Себастьян Дашнер
Java champion—2016

Изучаем Java EE

СОВРЕМЕННОЕ ПРОГРАММИРОВАНИЕ ДЛЯ БОЛЬШИХ ПРЕДПРИЯТИЙ

Packt>

ПИТЕР®

Architecting Modern Java EE Applications

Designing lightweight, business-oriented enterprise applications in the age of cloud, containers, and Java EE 8

Sebastian Daschner

BIRMINGHAM - MUMBAI

Себастьян Дашнер
Java champion—2016

Изучаем Java EE

СОВРЕМЕННОЕ ПРОГРАММИРОВАНИЕ ДЛЯ БОЛЬШИХ ПРЕДПРИЯТИЙ

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2018

Себастьян Дашнер

Изучаем Java EE. Современное программирование для больших предприятий

Серия «Для профессионалов»

Перевел с английского *А. Тумаркин*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художественный редактор	*С. Заматевская*
Корректор	*Е. Павлович*
Верстка	*Г. Блинов*

ББК 32.973.2-018.1 УДК 004.43

Дашнер С.

Д21 Изучаем Java EE. Современное программирование для больших предприятий. — СПб.: Питер, 2018. — 384 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0774-2

Java EE 8 — современная версия популярной платформы для программирования корпоративных приложений на языке Java. Новая версия платформы оптимизирована с учетом многочисленных технологических нововведений, среди которых — работа с контейнерами, улучшенные API для обеспечения безопасности, возможности работы с облачными хранилищами и микросервисной архитектурой. Java EE обеспечивает широкие возможности предметно-ориентированного проектирования (DDD), непрерывную интеграцию, работу по принципу DevOps, взаимодействие с Docker и Kubernetes.

Принципы проектирования и архитектурные секреты, собранные в этой книге в изложении великолепного Себастьяна Дашнера (в 2016 году удостоен звания Java-champion), послужат вам неисчерпаемым источником вдохновения и солидной базой для воплощения даже самой сложной бизнес-логики в Java-приложениях.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1788393850 англ.

ISBN 978-5-4461-0774-2

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 06.2018. Наименование: книжная продукция. Срок годности: не ограничен.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Подписано в печать 15.06.18. Формат 70x100/16. Бумага офсетная. Усл. п. л. 30,960. Тираж 1000. Заказ 0000.
Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Краткое содержание

Оглавление

Предисловие

В сфере разработки языки программирования сменяют друг друга, и темп, с которым создаются новые, кажется, только увеличивается. Ежегодно появляются новые языки, и за каждым — колоссальная работа по формированию целой экосистемы, актуальной до момента выхода нового языка. Любой тренд появляется и исчезает, и то же случается с экосистемой, созданной вокруг него.

Люди, вкладывающие в эти трендовые экосистемы значительные средства, часто сталкиваются с множеством проблем, которые приводят к неудачам в бизнесе и потере проектом импульса из-за невозможности найти и нанять разработчиков. По мере созревания экосистемы процесс распознавания и перестройки сложной системы зачастую приводит к труднорешаемым проблемам несовместимости. Рабочих инструментов оказывается недостаточно, или они содержат большое количество ошибок, а иногда и вовсе отсутствуют.

Источник силы экосистемы Java за последние 20 лет — ее стандарты, и главным из них является Java EE. Под эгидой Java EE собраны 53 запроса на спецификацию Java[1] (Java Specification Requests, JSR): от синтаксического анализа XML до синтаксического анализа JSON, от сервлетов до JAX-RS, от двоичных протоколов до протоколов RESTful, такие интерфейсные технологии, как JSF или MVC, API для маршалинга данных в XML (JAX-B) или JSON (JSON-B). Спецификации распространились столь широко, что даже если вы не рассматриваете себя как пользователя Java EE, даже если вы Java-разработчик, то все равно в определенной степени их применяете. По современным оценкам, в мире около 9 миллионов Java-разработчиков.

Область применения Java EE простирается от Walmart, крупнейшей в мире компании, занимающейся розничными продажами, и третьего по величине работодателя, до программы NASA SOFIA, исследующей космос на высоте 12 тысяч метров над Землей. Несмотря на то что сообщество разработчиков постоянно разрастается и корпорации все больше задействуют Java, современных ресурсов

---

[1] Формальные документы, описывающие спецификации и технологии, которые предлагается добавить к Java-платформе. — *Примеч. ред.*

Java EE невообразимо мало. Walmart и NASA, например, используют Apache TomEE, реализация которой занимает 35 Мбайт на диске, загружается за секунду и потребляет менее 30 Мбайт памяти. Эта сдержанность характерна для всех современных реализаций, включая WildFly, Payara и LibertyProfile. Инструменты Java EE, облака и IDE полны конкурентных решений, которые непросто отследить. Состоящая из более чем 200 человек компания ZeroTurnaround, например, работает на базе программы, добавляющей решения мгновенного развертывания на серверы Java EE.

Из-за того что экосистема столь обширна и имеет почти 20-летнюю историю, не так просто понять, что именно делает Java EE отличным решением сегодня и как его применять на практике в современном мире микросервисов. Очень легко найти подробную информацию двух-, пяти- или десятилетней давности. Статья, написанная авторитетным автором в одно время, может противоречить столь же авторитетному, но полностью противоположному видению автора статьи, созданной в другое время. Фактически у Java EE уникальная история — немногие технологии существовали столь долго и эволюционировали столь значительно.

Эта книга описывает новое поколение Java EE. Вы отправитесь в путешествие по Java EE в контексте современного мира микросервисов и контейнеров. Это скорее не справочное руководство по синтаксису API — изложенные здесь концепции и методики отражают реальный опыт человека, который сам недавно прошел этот путь, обращая пристальное внимание на возникающие препятствия, и готов поделиться своими знаниями. В различных ситуациях, начиная с создания пакета для тестирования и облачного использования, эта книга станет идеальным компаньоном и для начинающих, и для опытных разработчиков, стремящихся понять больше, чем просто API, и поможет им перестроить свое мышление для создания архитектуры современных приложений в Java EE.

Дэвид Блевинс (David Blevins),
основатель и CEO Tomitribe

Об авторе

Себастьян Дашнер (Sebastian Daschner) — Java-фрилансер, работающий консультантом и преподавателем, энтузиаст программирования и Java (EE). Он принимает участие в JCP, помогая создавать новые стандарты Java EE, обслуживая в JSR экспертные группы 370 и 374 и работая в различных проектах с открытым исходным кодом. За свой вклад в сообщество и экосистему Java он был награжден титулами чемпиона Java и чемпиона-разработчика Oracle.

Себастьян регулярно выступает на таких международных IT-конференциях, как JavaLand, JavaOne и Jfokus. Он получил награду JavaOne Rockstar на конференции JavaOne 2016. Вместе с менеджером сообщества Java Стивом Чином (Steve Chin) он посетил десятки конференций и пользовательских групп Java, путешествуя на мотоцикле. Стив и Себастьян создали JOnsen — неконференцию Java, которая проходила у термального источника в сельской местности в Японии.

О рецензенте

Мелисса Маккей (Melissa McKay) — разработчик программного обеспечения с 15-летним опытом создания приложений различных типов как для частных клиентов, так и для предприятий. Сейчас она занимается в основном серверными Java-приложениями, которые используются в области коммуникаций и телевидения. В сферу ее интересов входят кластерные системы, особую страсть она питает к решению задач, связанных с параллельной и многопоточной работой приложений.

Мелисса регулярно посещает неконференции JCrete на Крите (Греция) и с удовольствием приняла участие в открытии неконференции JOnsen в Японии. Она любит участвовать в волонтерских IT-конференциях для детей, например JavaOne4Kids и JCrete4Kids. Была членом комитета по контенту на JavaOne 2017 и является активным членом Denver Java User Group.

Благодарности

В процессе своей карьеры я имел честь познакомиться со многими людьми, оказавшими большое влияние не только на мою работу, но и на эту книгу, и есть немало тех, без которых она не была бы написана. Этот список увеличивается с каждым годом. Все эти люди косвенно помогли сформировать книгу, и я ценю каждого из них.

Есть несколько друзей, напрямую повлиявших на мою работу. Хочу поблагодарить отдельно:

- Керка Пеппердина (Kirk Pepperdine) — за его неутомимое стремление к развенчанию мифов в мире производительности программного обеспечения и разрешение использовать jPDM. Его бесценный опыт очень помог улучшить качество этой книги;
- Мелиссу Маккей (Melissa McKay) — за ее неустанную доработку текста, значительно улучшившую качество книги, за желание поделиться своим опытом в Enterprise Java и не в последнюю очередь — за вдохновение и мотивацию;
- Дэвида Блевинса (David Blevins) — за то, что он разделяет мою страсть к Java EE и написал предисловие к этой книге;
- Энди Гумбрехта (Andy Gumbrecht) — за помощь не только с темой корпоративного тестирования, но и с английским языком;
- Маркуса Эйзеле (Markus Eisele) — за то, что зажег искру этой работы;
- Филиппа Браненберга (Philipp Brunenberg) — за конструктивные вдохновляющие идеи и не в последнюю очередь за постоянную мотивацию в процессе написания книги.

Благодарю Кирин Бен Айед
за понимание, постоянную поддержку
и бесконечное терпение. Эта книга
посвящена тебе.

Введение

Java EE 8 предлагает множество функций, в основном ориентированных на новые архитектуры, такие как микросервисы, модернизированные API безопасности и облачные развертывания. Эта книга научит вас проектировать и разрабатывать современные бизнес-ориентированные приложения с использованием Java EE 8. В ней рассказывается, как структурировать системы и приложения и как шаблоны проектирования и технологии проблемно-ориентированного проектирования реализуются в эпоху Java EE 8. Вы изучите концепции и принципы, лежащие в основе приложений Java EE, и узнаете, как они влияют на коммуникацию, сохраняемость, техническую и сквозную функциональность, а также на асинхронное поведение.

В издании основное внимание уделяется реализации бизнес-логики и тому, как удовлетворить потребности клиентов в корпоративной среде. Рассказывается о том, как создавать промышленные приложения, применяя разумные технологические решения, без чрезмерного технического усложнения. В книге не только демонстрируется, как реализовать определенное решение, но и объясняется, почему выбрано именно оно.

Прочитав эту книгу, вы изучите принципы современной Java EE и освоите способы реализации эффективных архитектурных решений. Кроме того, вы узнаете, как разработать корпоративное программное обеспечение в эпоху автоматизации, непрерывной поставки кода и облачных платформ.

Структура книги

Глава 1 «Вступление» знакомит с корпоративными приложениями Java EE и рассказывает, почему Java EE по-прежнему пользуется популярностью в современных системах.

Глава 2 «Проектирование и структурирование приложений Java Enterprise» на примерах показывает, как спроектировать структуру корпоративного при-

ложения. Подразумевается проектирование корпоративных приложений для определенных бизнес-сценариев.

Глава 3 «Внедрение современных приложений Java Enterprise» описывает, как внедрять современные приложения Java EE, и объясняет, почему этот выбор технологий актуален и сегодня.

Глава 4 «Облегченная Java EE» учит создавать несложные приложения Java EE небольшого размера и с минимальной зависимостью от сторонних разработок.

Глава 5 «Java EE в контейнерных и облачных средах» объясняет, как использовать преимущества контейнеров и современных сред, как интегрировать корпоративные приложения и как это помогает улучшать рабочие процессы разработки.

Глава 6 «Рабочие процессы создания приложений» рассказывает о ключевых моментах, обусловливающих создание быстрых конвейеров разработки и обеспечение высокого качества программного обеспечения, от непрерывной поставки до автоматизированного тестирования и DevOps.

Глава 7 «Тестирование» как следует из названия, охватывает тему тестирования. Она поможет вам обеспечить высокое качество разработки автоматического тестирования программного обеспечения.

Глава 8 «Микросервисы и системная архитектура» рассказывает об основных моментах проектирования систем в соответствии с условиями проекта и компании, о том, как создавать приложения и их интерфейсы и когда требуются микросервисные архитектуры.

Глава 9 «Мониторинг, производительность и журналирование» объясняет, почему традиционное журналирование приносит вред, как разобраться в проблемах производительности и контролировать рабочие и технические характеристики приложения.

Глава 10 «Безопасность» рассказывает о реализации и интеграции задач безопасности в современных средах.

В заключительной, 11-й главе кратко обобщается весь изложенный материал, а также описываются некоторые стандарты Java EE 8.

Что вам понадобится для этой книги

Для запуска и выполнения примеров кода, приведенных в книге, вам потребуются следующие инструменты, настроенные в вашей системе:

- NetBeans, IntelliJ или Eclipse IDE;
- GlassFish Server;
- Apache Maven;
- Docker;
- Jenkins;
- Gradle.

Для кого предназначено это издание

Книга предназначена для опытных разработчиков Java EE, которые стремятся стать архитекторами приложений корпоративного уровня, а также для разработчиков программного обеспечения (ПО), которые хотели бы использовать Java EE для создания эффективных проектов приложений.

Условные обозначения

В издании вы увидите текст, оформленный различными стилями. Ниже приводятся примеры стилей и объясняется, что означает все это форматирование.

В тексте кодовые слова, имена таблиц базы данных, папок, файлов, расширений файлов, путей, пользовательский ввод показаны следующим образом: «EJB аннотируется с помощью `@Startup`». URL-адреса и твиттер-дескрипторы выделены особым шрифтом.

Блоки кода выделены следующим образом:

```
@PreDestroy
public void closeClient() {
    client.close();
}
```

Когда нужно привлечь внимание к определенному участку блока кода, соответствующие строки или элементы выделяются полужирным шрифтом:

```
private Client client;
private List<WebTarget> targets;

@Resource
ManagedExecutorService mes;
```

Для упрощения и улучшения читаемости некоторые примеры кода сокращены. Операторы Java, например `import`, указаны только для новых типов, а несущественные для примера участки кода обозначаются многоточием (`...`).

Все, что вводится и выводится в командной строке, записано в таком виде:

```
mvn -v
```

Новые термины и *важные слова* выделены курсивом.

Загрузка примеров кода

Примеры кода для выполнения упражнений из этой книги доступны для скачивания по адресу https://github.com/PacktPublishing/Architecting-Modern-Java-EE-Applications. Для скачивания материалов выполните следующие шаги.

1. Нажмите кнопку Clone or Download (Клонировать или скачать).
2. На открывшейся панели щелкните на ссылке Download Zip (Скачать Zip).

После того как архив будет загружен, можете распаковать его в нужную папку, используя последнюю версию одной из нижеперечисленных программ:

- WinRAR или 7-Zip для Windows;
- Zipeg или iZip или UnRarX для macOS;
- 7-Zip или PeaZip для Linux.

Удачи!

1 Вступление

К современному корпоративному ПО предъявляется много новых требований. Уже недостаточно лишь разработать некий программный продукт и развернуть его на сервере приложений. Впрочем, этого никогда не было достаточно.

Новые требования к корпоративным системам

Сегодня мир меняется быстрее, чем когда-либо прежде. Поэтому способность *быстро изменяться* — одно из важнейших требований к современным ИТ-компаниям. Последние способны оперативно приспособиться к реальному миру и потребностям клиентов. Ожидаемое время выхода новых функций на рынок сократилось от нескольких лет или месяцев до недель и дней. Чтобы уложиться в такие сроки, компаниям необходимо не только вводить новые технологии или *выделять больше денег* на реализацию бизнес-логики, но и переосмысливать и реорганизовывать базовые принципы работы их ИТ.

Что в этом контексте означает *быстро меняться*? В чем это выражается? Какие методы и технологии этому способствуют?

Быстро меняться означает быстро адаптироваться к потребностям рынка и клиентов. Если требуется новая функция и она кажется многообещающей, сколько понадобится времени, чтобы она прошла путь от первоначальной идеи до готового инструмента в руках пользователя? Если нужна новая инфраструктура, сколько времени пройдет от принятия решения до получения работающего оборудования? И не будем забывать о том, что, кроме разработки программного обеспечения в требуемые сроки, нужен еще автоматизированный контроль качества, который гарантирует, что все будет работать как ожидалось и не нарушится существующая функциональность.

При разработке программного обеспечения основная часть этих вопросов касается непрерывного развертывания и автоматизации. Новое программное обеспечение должно быть разработано, протестировано и поставлено автоматическим, быстрым, надежным и воспроизводимым способом. Надежный автоматизирован-

ный процесс обеспечивает не только более быстрые изменения, но и в конечном счете более высокое качество. Автоматический контроль качества, например выполнение программных тестов, — часть технологического процесса. В современной разработке ПО непрерывное развертывание, автоматизация и надлежащее тестирование являются одними из наиболее важных принципов.

Самым узким местом в большинстве компаний всегда была инфраструктура. Небольшие компании часто пытаются построить новую инфраструктуру, имея ограниченный бюджет. Более крупным компаниям, как правило, не удается разработать быстрые и эффективные процессы. Проблема многих больших корпораций заключается не в бюджете, а в технологиях. Часто приходится ждать появления новой инфраструктуры в течение нескольких дней или даже недель из-за согласований и чрезмерно сложных процессов, которые технически могли бы быть выполнены за считаные минуты.

Именно поэтому так важны инфраструктура приложения и способ его разработки. В главе 5 мы рассмотрим современные облачные среды. Вы увидите, что на самом деле не имеет особого значения, используются облачные услуги или нет. Быстрые и продуктивные процессы можно реализовать и на локальном оборудовании. Гораздо важнее, чтобы они правильно работали и применялись соответствующие технологии.

Современная инфраструктура должна создаваться быстро и автоматически, быть воспроизводимой и надежной. Она должна без особых усилий адаптироваться к меняющимся требованиям. Для того чтобы соответствовать этим требованиям, инфраструктура должна быть описана в виде кода — либо как процедурный сценарий, либо как набор декларативных описаний. Позже мы увидим, как инфраструктура в виде кода влияет на процессы разработки программного обеспечения и какие технологии ее поддерживают.

Данные требования относятся и к рабочим группам. Команде разработчиков уже недостаточно *просто создать* программный продукт и передать его операционным командам, которые его запустят, а затем будут решать возникающие в процессе эксплуатации проблемы. Такая практика чревата натянутыми отношениями и перекладыванием на других ответственности за критические ошибки. Но ведь общая цель должна заключаться в том, чтобы поставлять программное обеспечение, полностью удовлетворяющее своему функциональному назначению. Описывая необходимую инфраструктуру и конфигурацию в виде кода, группы разработчиков и операционистов должны составлять единую команду, отвечающую за работу программного обеспечения в целом. Это называется *DevOps* — культура разработки и эксплуатации, нацеленная на ответственность всей команды разработчиков ПО за функционирование продукта. Все участники отвечают за то, что клиенты будут использовать надлежащее программное обеспечение. Это скорее организационная, чем техническая задача.

Технически для решения этих задач применяется непрерывное развертывание, а также методика, которую называют *12-факторным* подходом и ориентацией на *выполнение в облаке*. Облачный и 12-факторный подходы описывают, как должны разрабатываться современные корпоративные приложения. Они определяют

требования не только к разработке, но и к эксплуатации приложений. В этой книге мы изучим указанные подходы, современные облачные среды и то, как их поддерживает Java EE.

Современный способ построения корпоративных систем

Посмотрим, как разрабатываются корпоративные программные комплексы.

Стремясь удовлетворить потребности реальных клиентов, мы должны спросить себя: каково назначение приложения, которое мы намерены разработать? Прежде чем углубляться в технические детали, необходимо прояснить мотивы и цели создания корпоративных программных систем, иначе получится разработка ради разработки. К сожалению, так происходит слишком часто. Сосредоточив внимание на бизнес-логике и принципах *проблемно-ориентированного проектирования*, как это прекрасно описано в книге Эрика Эванса (Eric Evans), мы гарантируем, что создаваемое программное обеспечение будет отвечать корпоративным требованиям.

Только после того, как назначение и задачи приложения станут понятны всем заинтересованным сторонам, можно перейти к техническим деталям. Группы разработчиков должны выбирать те методики и технологии, которые позволят не только эффективно реализовывать бизнес-сценарии использования продукта, но и уменьшить объем работы и издержки. У разработчиков должна быть возможность сосредоточиться на бизнесе, а не на операционной среде или технологии. Хорошая операционная среда тихо и незаметно помогает реализовывать бизнес-логику, не требуя внимания разработчика.

Выбранная технология должна обеспечивать и максимально эффективные процессы разработки. Это означает не только автоматизацию и быстрое выполнение всех этапов, но и поддержку современной инфраструктуры, такой как контейнеры Linux. В главах 4 и 5 мы подробнее рассмотрим технические особенности современных сред и поговорим о том, как их поддерживает Java EE.

Значение Java EE для современных систем

Перейдем к платформе Java EE, поскольку она имеет прямое отношение к корпоративным системам и это основная тема книги.

Java EE и J2EE используются очень широко, особенно в крупных компаниях. Одно из их преимуществ заключается в том, что платформа состоит из стандартов, гарантирующих обратную совместимость с ранними версиями. Даже старые приложения J2EE в дальнейшем будут гарантированно функционировать. Это всегда было весомым аргументом для компаний, строящих долгосрочные планы. Приложения, разработанные на базе Java EE API, способны работать на всех серверах приложений Java EE. Независимые от производителя приложения

позволяют компаниям создавать перспективное ПО, не ограниченное рамками какого-либо конкретного продукта. Это оказалось разумным решением и в итоге привело к появлению индустриальной культуры, в которой стандарты улучшают ситуацию в целом.

По сравнению с миром J2EE в Java EE многое изменилось. Здесь совершенно иная модель программирования — более компактная и продуктивная. Радикальные изменения произошли вместе со сменой названия с J2EE на Java EE 5 и особенно на EE 6. В главе 3 мы рассмотрим современный способ разработки приложений на Java. Узнаем, какие архитектурные подходы и программные модели используются, и увидим, что благодаря платформе разработка становится гораздо эффективнее, чем прежде. Тогда вы поймете, почему Java EE является современным решением для разработки корпоративных приложений.

Сейчас донести эту информацию до отрасли — в сущности, скорее маркетинговая и политическая, чем техническая задача. Множество разработчиков и архитекторов все еще считают Java EE громоздким, тяжеловесным корпоративным решением эпохи J2EE, требующим много времени, усилий и XML. За *Enterprise JavaBeans (EJB)* и серверами приложений сохраняется очень плохая репутация. Именно этим вызвано предубеждение многих инженеров против этой технологии. По сравнению с другими корпоративными решениями у Java EE было довольно мало маркетинговых решений для разработчиков.

В главе 4 мы увидим, почему современная Java EE является одним из самых простых корпоративных решений. Вы узнаете, за счет чего она столь удобна и почему сейчас более актуальна, чем когда-либо, особенно в современных облачных и контейнерных средах. Влияние ИТ-индустрии на конкретную технологию очень важно для достижения ею успеха.

Компании выбирают Java EE в основном из-за надежности и обратной совместимости. Я предпочитаю Java EE из-за ее производительности и простоты использования. Подробнее об этом мы поговорим в главах 4 и 5. В этой книге я хотел бы доказать читателям, что Java EE — решение, хорошо удовлетворяющее потребности современных предприятий. Я также представлю вам технологии и стандарты — не углубляясь в подробности. Скорее расскажу, как они связаны между собой. Считаю, что, сосредоточившись на этой связи, мы лучше поймем, как эффективно строить промышленные приложения.

Обновление и перспектива развития Java EE 8

Сейчас мы кратко рассмотрим, что появилось в Java EE версии 8. Цель создания этой версии — сделать продукт еще удобнее для разработчика, оптимизировать работу API и привести Java EE в соответствие с новыми требованиями, предъявляемыми облачными средами. Платформа располагает двумя полностью новыми JSR — *JSON-B* (Java API для JSON Binding) и *Security*. Кроме того, усовершенствованы существующие стандарты. В частности, внедрение JSON-B упрощает независимую от поставщика интеграцию интерфейсов JSON HTTP API.

Назначение Java EE — улучшить разработку корпоративных приложений в соответствии с современными средами и условиями. Оказывается, современные среды не просто совместимы с Java EE — они поддерживают подходы, которые уже много лет являются частью платформы, например отделение API от реализации или мониторинг сервера приложений.

В долгосрочной перспективе предполагается оптимизировать поддержку современных методов мониторинга, проверки работоспособности и устойчивости. Сегодня для реализации каждой из этих возможностей приходится слегка дополнять код, в чем мы убедимся в следующих главах. Предполагается, что в дальнейшем такая интеграция упростится. Java EE организована так, чтобы разработчик мог сосредоточиться на своих прямых задачах — заняться реализацией бизнес-логики.

Java Community Process

Уникальность платформы Java EE — в процедуре ее определения. Стандарты Java EE разрабатываются как часть *Java Community Process (JCP)*. JCP — яркий пример отрасли, где активно поощряется участие всех, кто интересуется данной технологией. Платформа включает в себя стандарты в виде запросов на спецификацию *Java Specification Requests (JSR)*. Запросы JSR применимы не только в Java и Java EE, но и в отношении основанных на них технологий, таких как среда разработки Spring. В результате практический мировой опыт разработки этих технологий помогает формировать новые JSR.

При разработке приложений, особенно при возникновении потенциальных проблем, письменные спецификации, сформированные на основе JSR, оказываются чрезвычайно полезными. Поставщики, поддерживающие корпоративную платформу, обязаны обеспечить реализацию так, как указано в стандартах. Таким образом, в спецификационных документах содержится информация и для поставщиков, и для разработчиков о том, как будет работать технология. Если какие-либо функции не работают, поставщики должны исправлять эти проблемы в своих реализациях. Это также означает, что разработчикам теоретически остается только изучать и знать сами эти технологии, а не детали, зависящие от поставщика.

Любой разработчик может участвовать в Java Community Process, чтобы помочь сформировать будущие версии Java и Java EE. *Экспертные группы (expert groups)*, работающие над конкретными стандартами, приветствуют конструктивную обратную связь от любого, кто интересуется этой темой, даже если он не состоит в JCP. Более того, можно ознакомиться со следующими версиями стандартов еще до их выпуска. Все это привлекает разработчиков архитектуры и компаний: они не только понимают направление развития, но и имеют возможность повлиять на него и внести свой вклад в процесс.

Именно по этим соображениям я сам специализируюсь на Java EE. У меня есть опыт разработки корпоративных систем в среде Spring. Кроме того, что обе технологии очень похожи с точки зрения модели программирования, я особенно

ценю эффективность стандарта CDI, а также возможность легко использовать все технологии платформы. Я изучал конкретные JSR, которые являются частью корпоративной платформы, участвовал в разработке инструментов, которые были стандартизированы в то время. Кроме того, я являюсь членом двух экспертных групп — по JAX-RS 2.1 и JSON-P 1.1. Участвуя в определении этих стандартов, я значительно пополнил свои знания в области корпоративных систем. Конечно, придется глубоко изучать конкретную технологию, если вы захотите помогать ее стандартизировать. Но, согласитесь, приятно осознавать, что принимаешь участие в разработке стандарта ИТ-индустрии.

Что вы найдете в этой книге

Я решил написать книгу о том, что узнал сам, работая со всевозможными корпоративными системами Java. Моя цель — описать вам современную технологию Java EE. Она предназначена прежде всего для разработки корпоративных приложений и современных моделей программирования. Я хочу, чтобы вы хорошо понимали, как Java EE используется в эпоху EE 8 и где проявляются лучшие свойства платформы. На базе современных сред разработки появились новые шаблоны и парадигмы проектирования. Если вы знакомы с миром J2EE, то, надеюсь, оцените и преимущества современной Java EE. Я постарался показать, какие из старых парадигм, ограничений и соображений, из-за которых многие разработчики не любили J2EE, больше неактуальны. Кроме этого, попытаюсь заразить вас своим энтузиазмом и объяснить, почему я убежден в том, что Java Enterprise хорошо подходит для построения корпоративных приложений.

Вам не обязательно разбираться в шаблонах и рекомендованных методах разработки J2EE. В частности, новая модель программирования настолько отличается от старой, что я убежден: имеет смысл продемонстрировать современный подход с чистого листа.

Если вам приходилось разрабатывать J2EE-приложения, это отлично. Вы увидите, как трудно было решать многие задачи с помощью старых шаблонов проектирования J2EE, особенно в современном мире, когда можно сосредоточиться на бизнес-требованиях, а не на технологии, используемой для их реализации. Это тем более актуально, если следовать принципам проблемно-ориентированного проектирования. Вы заметите, скольких громоздких и утомительных методов, в прошлом свойственных системам J2EE, можно избежать в современной Java EE. Простота и мощь платформы Java EE могут вдохновить вас на переосмысление подходов, которые вы применяли до сих пор.

Эта книга предназначена для инженеров-программистов, разработчиков и архитекторов, которые создают корпоративные приложения. В издании я в основном буду использовать термины «разработчики» или «инженеры». Тем не менее я убежден, что архитекторам также время от времени стоит заглядывать в исходный код. Это нужно не только для поддержки других разработчиков в команде — это важно для них самих, ведь так можно получить больше практического опыта.

Точно так же любой разработчик должен иметь по крайней мере базовое представление о системной архитектуре и понимать причины, по которым принимаются определенные решения. Чем лучше специалисты станут понимать друг друга, тем проще им будет общаться и тем успешнее пройдет работа над проектом.

Разработка современных корпоративных приложений — это гораздо больше, чем просто разработка. Это и новые требования, предъявляемые к корпоративным приложениям, и инженеры, которые вникают в процессы, и облачные среды, контейнеры и инструменты управления ими. Мы подробно изучим проблемы непрерывной поставки ПО и автоматизированного тестирования, поговорим о том, почему они имеют такое большое значение и как интегрируются с Java EE. Мы также рассмотрим контейнерные технологии, такие как Docker, и среды управления, например Kubernetes. В современном мире корпоративных систем важно показать, как технология наподобие Java EE поддерживает эти возможности.

Архитектура Microservice — обширная тема, еще одна из сегодняшних проблем. Мы разберем, что такое микросервисы и как они могут быть реализованы с помощью Java EE. Кроме того, обязательно поговорим о безопасности, протоколировании, производительности и мониторинге. Я покажу, что именно должен знать и принимать во внимание архитектор, работая над современным ПО. В частности, ему может потребоваться определить набор нужных технологий, особенно когда речь идет о современных решениях, например, в области 12-факторных или облачных приложений. Однако гораздо важнее понимать, какие концепции лежат в основе этих технологий и каково их назначение. Используемая технология меняется с каждым днем, принципы и концепции информатики живут гораздо дольше.

Мой подход ко всем темам, затронутым в этой книге, заключается в том, чтобы сначала объяснить причины, лежащие в основе принятых решений, а затем показать, как эти решения применяются и внедряются в Java EE. Я считаю, что простое изучение определенной технологии, безусловно, может помочь разработчикам в их повседневной работе, но они не усвоят материал полностью до тех пор, пока не поймут до конца, в чем назначение того или иного подхода. Вот почему я начну с описания корпоративных приложений в целом.

В Java EE много функций, особенно если сравнивать с прошлыми версиями. Но эта книга не является полным справочником по Java EE. При ее написании я стремился рассказать вам о своем практическом опыте, а также дать *практические рекомендации* с подробным разбором решений типичных сценариев. Итак, приготовьтесь насладиться путешествием в мир современного корпоративного программного обеспечения.

2 Проектирование и структурирование приложений Java Enterprise

Любая часть программного обеспечения спроектирована по конкретной схеме. Проектирование охватывает архитектуру системы, структуру проектов, структуру и качество кода. Оно может либо ясно сообщать о назначении системы, либо вводить в заблуждение. Прежде чем разработать корпоративное приложение или систему, инженеры должны их спроектировать. Для этого необходимо определить назначение данного программного обеспечения (ПО).

В этой главе будут рассмотрены следующие темы.

- Чему нужно уделять особое внимание при разработке ПО.
- Структуры сборки проекта и системы сборки Java EE.
- Как структурировать модули корпоративных проектов.
- Как реализовать пакетные структуры модулей.

Назначение корпоративных приложений

У каждого действия как в повседневной жизни, так и в деятельности крупной организации или разработке программного проекта, есть своя причина. Нам, людям, нужны причины, для того чтобы что-то делать. Разработка корпоративных программных продуктов не исключение.

При создании программного приложения в первую очередь нужно задать вопрос: *зачем*? Зачем нужна эта часть программы? Почему необходимо потратить время и силы на разработку данного решения? И почему компания должна сама заботиться о разработке этого решения?

Другими словами: каково назначение приложения? Какую проблему должно решать это программное обеспечение? Хотим ли мы, чтобы приложение выполняло важный бизнес-процесс? Принесет ли это прибыль? Будет ли оно приносить

доход непосредственно, например путем продажи товаров, или косвенно — за счет маркетинга, поддержки клиентов или бизнес-процессов? Существуют ли другие возможности обслуживания клиентов, работников или бизнес-процессов?

Эти и другие вопросы определяют бизнес-цели приложения. Вообще говоря, каждый элемент программного обеспечения нуждается в таком обосновании, чтобы сначала увидеть общую картину и только потом вкладывать в разработку время и силы.

Наиболее очевидным обоснованием является реализация необходимых бизнес-сценариев. Они имеют определенную ценность для всего бизнеса, рано или поздно автоматизируются и приносят доход. В конце концов, главной целью создания ПО является максимально эффективная реализация бизнес-сценариев.

На чем сосредоточиться разработчику

Разработчикам программного обеспечения, а также руководителям проектов следует в первую очередь ориентироваться на требования предприятия и сосредоточиться на реализации бизнес-сценариев.

Это кажется очевидным, но слишком часто внимание в корпоративных проектах уделяется другим проблемам. Разработчики тратят время и силы на реализацию деталей и функций, бесполезных при решении актуальных задач. Вспомните, сколько реализаций журналирования, самодельных корпоративных фреймворков и технически перегруженных уровней абстракций встречалось нам за последнее время.

Нефункциональные требования, качество ПО и так называемые сквозные проблемы на самом деле являются важными составляющими разработки ПО. Но первой и главной точкой приложения всех инженерных усилий должны быть удовлетворение требований бизнеса и разработка программного обеспечения, которое действительно приносит пользу.

Удовлетворение требований клиентов

Ответьте на следующие вопросы.

- В чем назначение приложения?
- Какие его функции наиболее важны для пользователей?
- Будет ли приложение приносить доход?

Ответы на эти вопросы должны быть известны всем заинтересованным сторонам. Если это не так, то следует сделать шаг назад, еще раз проанализировать проект в целом и пересмотреть его право на существование. Не всегда цель создания программного обеспечения диктуется исключительно бизнесом. На практике во многих случаях внедряют решения, которые не приносят дохода напрямую, но делают это косвенно, поддерживая другие продукты. Такие решения, безусловно,

необходимы, и мы рассмотрим их в главе 8 в рамках вопроса о том, как создать рациональную системную среду.

За исключением этих вспомогательных программных систем мы будем концентрироваться на бизнес-аспектах. Помня об этой главной цели, первое, на что следует обратить внимание, — способ моделирования бизнес-сценариев и преобразования их в программное обеспечение. Только после этого можно реализовывать сценарии с помощью тех или иных технологий.

Такие приоритеты отражают также требования клиентов. Все участники разработки приложения заинтересованы в том, чтобы оно соответствовало своему назначению.

Инженеры-разработчики программного обеспечения склонны видеть все иначе. Они заботятся о деталях реализации и изяществе решений. Инженеры часто увлечены той или иной технологией и тратят много времени и усилий на выбор правильного решения и его качественную реализацию. Это влечет за собой множество сквозных технических проблем, таких как ведение журнала и чрезмерное техническое усложнение системы, что не является обязательным с точки зрения бизнеса. Высокое мастерство программирования, безусловно, имеет большое значение и необходимо для написания хорошего программного обеспечения, но во многих случаях оно расходится с целями клиента. Прежде чем тратить время и усилия на реализацию технических деталей, инженеры должны изучить требования клиента.

Требования к срокам выполнения проекта — еще один аспект, который необходимо учесть. Группы разработчиков оценивают бизнес-сценарии, сравнивая их с качеством технических решений. Они склонны отказываться от необходимых тестов ПО и проверки качества, чтобы уложиться в сроки. Технология, используемая для внедрения бизнес-приложения, должна поддерживать эффективную и прагматичную разработку.

Посмотрев на корпоративный мир глазами клиента, который платит деньги, или менеджера с ограниченными временем и бюджетом, разработчики программного обеспечения, скорее всего, поймут их приоритеты. Первостепенную важность приобретает реализация сценариев, приносящих доход. Техническое обеспечение, выходящее за рамки этого, рассматривается клиентами и менеджерами как бесполезное.

В дальнейшем будет показано, как соединить и сбалансировать эти две мотивации с помощью Java EE.

Внешняя структура корпоративного проекта

Помня о главной цели — реализации бизнес-сценариев, — спустимся с небес на землю и обратим внимание на реальные корпоративные проекты. В следующих главах мы узнаем, какие методы помогут нам правильно описать бизнес в архитектуре приложения.

Структура бизнеса и группы разработчиков

Программные проекты обычно разрабатывает команда инженеров, разработчиков ПО или архитекторов. Для простоты мы будем называть их программистами. Не ошибусь, если скажу, что всем им — разработчикам ПО, архитекторам, тестировщикам и другим инженерам — время от времени приходится программировать.

Почти всегда над программным проектом одновременно работают несколько человек. Поэтому требуется учесть еще пару моментов, главным образом коммуникационные и организационные издержки. Если же рассмотреть структуру организации, состоящей из нескольких команд, работающих над несколькими проектами или даже — временно — над одним проектом, то мы столкнемся с еще бóльшими проблемами.

Закон Конвея гласит: *«Организации, проектирующие системы <...> вынуждены создавать проекты, которые являются копиями коммуникационных структур этих организаций»*.

Другими словами, способ организации команд и коммуникации между ними неизбежно отражается на структуре ПО. При разработке программных проектов следует учитывать организационную схему, в соответствии с которой работают программисты, и их эффективные коммуникационные структуры. О том, как создать несколько распределенных систем и конкретные микросервисы, я подробно расскажу в главе 8.

Даже в одном проекте, выполняемом небольшой командой программистов, скорее всего, будет создаваться несколько функций или исправляться несколько ошибок одновременно. Это обстоятельство влияет на то, как мы планируем итерации, строим и интегрируем исходный код, а также создаем и развертываем готовое программное обеспечение. Данные вопросы будут рассмотрены, в частности, в главах 6 и 7.

Содержимое программных проектов

Проект корпоративного программного обеспечения включает в себя несколько рабочих продуктов, необходимых для создания и распространения приложений. Рассмотрим их подробнее.

Исходный код приложения

Все корпоративные приложения, как и любые другие, записываются в виде исходного кода. Исходный код, вероятно, является самой важной частью программного проекта. В нем содержится само приложение и определяются все его функциональные возможности. Он может рассматриваться как единственный источник достоверных данных, определяющих поведение программного обеспечения.

Весь исходный код проекта можно разделить на производственный код для работы готового продукта и тестовый код, необходимый для того, чтобы проверить поведение приложения. Тестовому и производственному коду соответствуют

различные технологии, к их качеству предъявляются разные требования. Технологии и структуры программных тестов мы подробно рассмотрим в главе 7. В остальных главах станем говорить о производственном коде, который поставляется клиентам и обрабатывает бизнес-логику.

Программные структуры

Исходный код программного проекта упорядочен в виде определенных структур. В Java-проектах есть возможность объединять компоненты и задачи в Java-пакеты и модули проекта (рис. 2.1).

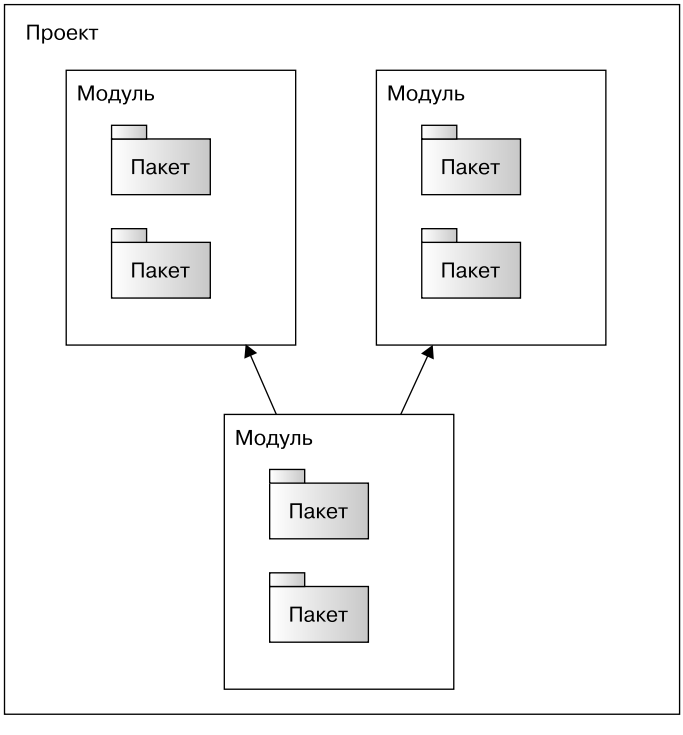

Рис. 2.1

Структурирование этих компонентов — не столько техническая, сколько архитектурная необходимость. С технической точки зрения код, разбитый на пакеты произвольно, тоже будет работать. Однако структура помогает инженерам лучше понять программное обеспечение и реализуемые им задачи. Объединяя в группы программные компоненты, выполняющие согласованные функции, мы увеличиваем согласованность и улучшаем структурирование исходного кода.

В этой и следующей главах мы обсудим преимущества проблемно-ориентированного проектирования, описанного в книге Эрика Эванса (Eric Evans), и узнаем, почему и как следует разбивать код на пакеты в соответствии с бизнес-логикой.

Пока просто запомним, что мы группируем согласованные компоненты, так что логические функции будут объединяться в логические пакеты или модули проекта.

Java SE 9 поставляется с возможностью установки модулей в формате Java 9. Эти модули похожи на JAR-файлы с возможностью декларации зависимостей и использования ограничений других модулей. Поскольку эта книга посвящена Java EE 8, а модули Java 9 еще не получили широкого распространения в реальных проектах, рассмотрим только пакеты Java и модули проекта.

Следующим элементом в структуре программных проектов является более мелкая единица программных компонентов — класс Java. Классы и их функции заключают в себе определенный функционал. В идеале они слабо связаны между собой и отличаются высокой целостностью.

Есть много литературы о практиках чистого кода и представлении функциональности в исходном коде. Например, в книге «Чистый код»[1] (Clean Code) Роберта К. Мартина (Robert C. Martin) описаны такие методы, как правильное именование и рефакторинг, которые позволяют получить добротный исходный код в виде пакетов, классов и методов.

Системы контроля версий

Поскольку большинство программных проектов требуют координации при изменении кода, выполняемом одновременно несколькими программистами, исходный код хранится в системе контроля версий. *Системы контроля версий (version control systems, VCS)* зарекомендовали себя как обязательные средства, необходимые для надежной координации и отслеживания изменений в программных системах. Они же помогают понять, в чем заключаются эти изменения.

Существует множество вариантов систем контроля версий, таких как Git, Subversion, Mercurial и CVS. За последние годы *распределенные системы контроля версий*, в частности *Git*, были широко признаны самыми современными инструментами. Для хранения и разрешения конфликтов между отдельными версиями в них используется так называемое хеш-дерево, или дерево Меркля (Merkle tree), что обеспечивает эффективные различия и слияния.

Распределенная VCS позволяет разработчикам работать с репозиториями проекта в распределенной среде, не требуя постоянного подключения к сети. Каждая рабочая станция имеет собственный репозиторий, который включает в себя полную историю изменений и периодически синхронизируется с центральным репозиторием проекта.

На момент написания этой книги в подавляющем большинстве программных проектов для контроля версий применяли систему Git.

Исполняемый код

В VCS-репозитории проекта хранится только исходный код, который создается и поддерживается разработчиками. Разумеется, корпоративные приложения

[1] *Мартин Р.* Чистый код. Создание, анализ и рефакторинг. — СПб.: Питер, 2018.

развертываются в виде своего рода двоичных артефактов. Только эти готовые двоичные файлы могут выполняться в виде исполняемого программного кода. Именно двоичные файлы в конечном счете и являются результатом процесса разработки и компиляции приложения.

В мире Java это означает, что исходный код Java компилируется в переносимый байт-код и обычно упаковывается в виде *архива веб-приложений* (*Web Application Archive, WAR*, или *Java Archive, JAR*, соответственно). WAR- и JAR-файлы содержат все классы и файлы, структурные зависимости и библиотеки, необходимые для установки приложения. В итоге *виртуальная машина Java (Java Virtual Machine, JVM)* выполняет байт-код, а вместе с этим и бизнес-функции.

В корпоративных проектах артефакты развертывания, представляющие собой WAR- или JAR-файлы, либо развертываются в контейнере приложений, либо предоставляют сам контейнер. Контейнер приложений необходим, поскольку, помимо реализации собственно бизнес-логики, корпоративные приложения решают дополнительные задачи, такие как поддержка жизненного цикла приложения или обеспечение всевозможных форм коммуникации. Например, веб-приложение, которое реализует определенную логику, но не передает данные по HTTP, вряд ли будет полезным. В Java Enterprise за такую интеграцию отвечает контейнер приложения. Упакованное приложение содержит только бизнес-логику и развертывается на сервере, который заботится обо всем остальном.

За последние годы появилось много Linux-технологий, таких как Docker, что способствует распространению идеи готовых двоичных файлов. Такие двоичные файлы содержат не только упакованные Java-приложения, но и все компоненты для его запуска: сервер приложений, виртуальную машину Java и необходимые двоичные файлы операционной системы. В главе 4 мы обсудим вопросы установки и развертывания корпоративных приложений, особенно в отношении контейнерных технологий.

Двоичные файлы создаются в процессе сборки программного обеспечения, что позволяет надежно воссоздать все двоичные файлы из исходного кода, хранящегося в репозитории. Поэтому двоичные файлы не должны храниться в системе контроля версий. То же самое касается и сгенерированного исходного кода. Раньше, например, классы JAX-WS, необходимые для SOAP-коммуникации, обычно генерировались из файлов дескрипторов. Генерируемый исходный код — созданный в процессе сборки, он не должен храниться в системе контроля версий. Хранить в репозитории следует только чистый исходный код, но не получаемые из него рабочие продукты.

Системы сборки

Процесс сборки в первую очередь обеспечивает компиляцию исходного кода программного проекта Java в байт-код. Это происходит каждый раз, когда в проект вносят изменения. Все современные системы сборки поставляют с удобными начальными установками, чтобы свести к минимуму дополнительную настройку.

В корпоративном мире с его многочисленными структурами и библиотеками важными этапами являются организация и определение всех зависимостей от

API и реализаций. Системы сборки, такие как *Apache Maven* и *Gradle*, упростили разработчикам жизнь, предоставив им эффективные механизмы разрешения зависимостей. Система сборки записывает все зависимости (с соответствующими версиями), необходимые для компиляции и запуска приложения. Это упрощает настройку проекта для нескольких разработчиков, а также позволяет создавать повторяемые сборки.

Упаковка скомпилированных классов и их зависимостей в артефакты развертывания также является частью процесса сборки. В зависимости от используемой технологии артефакты упаковываются в WAR- или JAR-файлы. В главе 4 мы рассмотрим различные способы упаковки корпоративных приложений Java, их преимущества и недостатки.

Далее, в разделах «Gradle» и «Apache Maven», мы подробно обсудим реализацию и различия этих двух основных систем сборки.

Одно- и многомодульные проекты

Как уже говорилось, исходный код приложения можно объединять в Java-пакеты и модули проекта. Модули, построенные по принципу общей функциональности, представляют собой отдельно компилируемые подпроекты. Обычно они определяются системами сборки.

На первый взгляд, причины разделения проекта на модули вполне понятны. Объединение Java-кода и пакетов во взаимосвязанные модули позволяет создать более четкое представление для разработчиков, обеспечивает лучшую структуру и повышает согласованность.

Еще одна причина построения многомодульной структуры — это скорость сборки. Чем сложнее программный проект, тем дольше он будет компилироваться и упаковываться в артефакт. Разработчики, как правило, в каждый момент времени затрагивают лишь отдельные части проекта. Поэтому идея состоит в том, чтобы каждый раз перестраивать не весь проект, а только модули, необходимые для внесения желаемых изменений. Именно это преимущество предоставляет система сборки Gradle, обещая сэкономить время за счет того, что перестраиваются только те части, которые были изменены.

Еще одним аргументом в пользу этой практики является возможность повторного использования подмодулей в других проектах. Создавая подпроекты и выделяя их в самостоятельные артефакты, теоретически можно взять субартефакт из одного проекта и включить в другой. Например, обычным делом является создание модуля *модели*, который содержит все сущности, относящиеся к области бизнеса, обычно в виде *автономных простых объектов Java (plain old Java objects, POJO)*. Такая модель упаковывается в JAR-файл и повторно задействуется как зависимость в других корпоративных проектах.

Однако у этой практики есть ряд недостатков, или, скорее, она внушает определенные иллюзии.

Иллюзии повторного использования

Напомню: программное обеспечение создает команда разработчиков, и структура проекта соответствует их коммуникационным структурам. Поэтому повторное применение модулей в рамках нескольких проектов требует некоторой координации.

Технические зависимости

Модуль проекта, предназначенный для повторного использования, должен соответствовать определенным критериям. Прежде всего технология общих модулей должна соответствовать новому проекту. Это кажется очевидным, поскольку влияет на детали реализации. Особенно сильно связаны с подключенными модулями и зависят от конкретной технологии применяемые библиотеки и структуры. Например, классы моделей в Java EE обычно содержат аннотации от API, такие как JPA, которые должны быть доступны во всех зависимых модулях.

Еще важнее с технической точки зрения зависимости от продуктов определенных версий сторонних производителей, необходимые для правильного функционирования совместно используемого модуля. Эти зависимости должны быть доступны при выполнении и не должны вызывать конфликтов с другими зависимостями или версиями. Коллизии с уже доступными на сервере зависимостями могут вызвать массу проблем. То же самое относится к деталям реализации, которые содержат неявные зависимости.

Типичным примером, иллюстрирующим сказанное, являются библиотеки отображения JSON, такие как Jackson и Gson. Многие сторонние зависимости задействуют определенные версии этих библиотек, которые могут вызывать конфликты с другими зависимостями или версиями при выполнении приложения. Другим примером являются реализации журналирования, такие как *Logback* и *Log4j*.

В целом совместно используемые модели должны быть как можно более самодостаточными или хотя бы содержать стабильные зависимости, которые не вызовут описанных проблем. Хорошим примером очень стабильной зависимости является Java EE API. Благодаря обратной совместимости с Enterprise Edition применение этого API и обеспечиваемая им функциональность не будут нарушены при появлении новой версии.

Но даже если Java EE API является единственной зависимостью общих модулей, это привязывает модель к определенной версии и уменьшает свободу внесения изменений.

Организационные проблемы

Совместно используемые технологии и зависимости вызывают ряд организационных проблем. Чем больше количество программистов и рабочих групп, тем сильнее влияние таких технологий и зависимостей. Группы должны согласовывать между собой применение определенных технологий, фреймворков, библиотек и их версий.

Если какая-то группа захочет что-то изменить в этой системе зависимостей или технологий, то такое изменение потребует значительной координации и накладных расходов. Вопросы совместного использования кода и артефактов в нескольких системах, а также их целесообразность рассматриваются в главе 8.

Критерии повторного использования

Повторное использование — это всегда компромисс между простотой и возможным дублированием. Выбор в пользу того или другого определяется уровнем самодостаточности. Как правило, затраты на координацию зависимостей, версий и технологий перевешивают преимущества, полученные за счет избегания избыточности.

Однако остается важный вопрос: какие уровни образуют модули проектов по вертикали и горизонтали? Примером горизонтальной структуры является типичная трехуровневая архитектура кластеризации в виде представления, бизнес-логики и данных. Вертикальное разделение на уровни означает группировку функционала на основе бизнес-логики. Примерами могут быть модули учетных записей, заказов или статей, включающие в себя все технические требования, такие как конечные точки HTTP и доступ к базе данных. Теоретически оба типа модулей могут быть задействованы многократно.

Выше вероятность использования в других проектах горизонтальных многоуровневых модульных структур, таких как модели. Естественно, у модулей этого типа меньше зависимостей, в идеале их вообще нет. Вертикальные многоуровневые модульные структуры, наоборот, содержат детали реализации и ожидают определенных условий, таких как заданная конфигурация контейнера. Но это также зависит от технологии, используемой в рамках модулей, которые должны применяться многократно.

Артефакты проекта

Теперь немного отвлечемся и переключимся на артефакты развертывания нашего корпоративного приложения. Как правило, приложение создает один артефакт, который и будет запускать программное обеспечение. Даже многомодульный проект в итоге сводится к одному-двум артефактам. Таким образом, в большинстве случаев вся эта структура снова сводится к одному JAR- или WAR-файлу. С учетом того, что повторно использовать модули не всегда возможно, возникает вопрос: так ли нужна многомодульность в проекте? В конце концов, создание и управление подпроектами, вертикальными или горизонтальными, потребует от программистов определенных усилий.

Действительно, разделение кода способно ускорить сборку, если перестраиваются только те подпроекты, в которые вносились изменения. Однако в разделах «Apache Maven» и «Gradle», а также в главе 4 мы увидим, что сборка единого разумно построенного проекта в один артефакт выполняется довольно быстро и обычно есть другие причины, влияющие на скорость создания сборок.

Один проект — один артефакт

Целесообразно упаковать проект корпоративного продукта в один артефакт развертывания, создаваемый из единственного модуля проекта. Количество и структура артефактов развертывания соответствуют структуре программного проекта. Если в проекте появляются дополнительные артефакты, то они представлены в виде отдельных модулей. Это позволяет сформировать понятную и простую структуру проекта.

Обычно в корпоративном проекте создается переносимый JAR- или WAR-файл, генерируемый из одного модуля проекта. Однако иногда есть веские причины для разработки модулей, применяемых в нескольких проектах. Они, что вполне логично, оформляются в виде отдельных проектных модулей, создающих собственные артефакты, например в виде JAR-файлов.

Есть и другие причины для появления многомодульных проектов. Системные тесты, проверяющие внешнее поведение развернутого корпоративного приложения, не всегда зависят от кода готового продукта. В некоторых ситуациях имеет смысл представить эти тесты в виде отдельных модулей многомодульного проекта.

Другим примером являются клиентские технологии, которые попросту слабо связаны с серверной частью приложения. При использовании современных клиенториентированных фреймворков JavaScript связь с серверной частью также ослабевает. Рабочий процесс и жизненный цикл разработки клиентской части могут отличаться от таковых для серверной части приложения. Поэтому имеет смысл разделить технологию на несколько подпроектов или даже на несколько программных проектов. Подробнее мы обсудим эту тему в разделе «Структурирование для современных клиентских технологий».

Однако эти ситуации в более широком смысле также отвечают концепции соответствия артефактов модулям проекта. Проект тестирования системы используется и выполняется отдельно от кода готового продукта. Разработка и построение клиентской части тоже могут отличаться от разработки и построения серверной. Бывают и другие ситуации, когда такое разделение целесообразно.

Сборка систем в Java EE

Модули проекта описаны в виде модулей системы сборки. Независимо от того, сколько в системе проектов — один или несколько, например код продукта и системные тесты, — эти части строятся и выполняются как часть процесса сборки.

Хорошая система сборки должна иметь определенные функции. Ее основная задача состоит в том, чтобы скомпилировать исходные коды и упаковать двоичные файлы в виде артефактов. На этапе компиляции или упаковки также добавляются все необходимые зависимости. Есть несколько областей, где требуются такие зависимости, — например на этапе компиляции, тестирования или реализации. На каждом этапе нужны свои зависимости, включаемые в артефакт.

Сборка проекта должна быть надежной и воспроизводимой. Несколько сборок одного и того же исходного проекта с одинаковой конфигурацией должны давать одинаковые результаты. Это важно для реализации конвейеров *непрерывной поставки (Continuous Delivery, CD)*, позволяющих создавать воспроизводимые сборки. При этом система сборки должна запускаться на сервере *непрерывной интеграции (Continuous Integration, CI)*, таком как *Jenkins* или *TeamCity*. Для этого нужно, чтобы программное обеспечение управлялось через интерфейс командной строки, особенно в системах на основе Unix. Причины необходимости непрерывной поставки будут описаны в главе 6.

Система сборки, используемая инженерами-программистами, должна поддерживать те фреймворки и операционные системы, в которых они работают. Для систем сборки на основе JVM такая переносимость обычно предусмотрена. Иногда проекты имеют особые требования, такие как необходимость построения собственного кода для какого-то конкретного окружения. Однако в корпоративных приложениях Java это обычно не требуется.

В целом процесс сборки должен выполняться как можно быстрее. Загрузка и настройка системы сборки не должны быть продолжительными. Чем больше времени требует сборка, тем дольше придется ждать инженерам обратной связи в процессе разработки. Подробнее мы рассмотрим эту тему в главе 4.

На момент написания книги самой распространенной системой сборки была известная большинству Java-программистов Apache Maven.

Maven — это система сборки на базе Java, настраиваемая посредством XML. Проекты в ней определяются в рамках так называемой *объектной модели проекта (project object model, POM)*. Maven основана на принципе программирования по соглашениям, что сводит к минимуму дополнительную настройку. Стандартная конфигурация хорошо подходит для Java-приложений.

Еще один распространенный инструмент сборки — Gradle. Он включает в себя основанный на Groovy *предметно-ориентированный язык (Domain-Specific Language, DSL)*, предназначенный для настройки полностью расширяемых сборок проектов с поддержкой сценариев. Поскольку Groovy является полноценным языком программирования, сценарии сборки Gradle, естественно, являются эффективными и гибкими.

И Gradle, и Maven включают в себя расширенное управление зависимостями и хорошо подходят для сборки Java-проектов. Конечно, есть и другие системы сборки, такие как SBT, но Gradle и Maven являются наиболее распространенными и будут рассмотрены в следующих разделах.

Apache Maven

Система сборки Apache Maven широко используется в Java-проектах и известна подавляющему большинству разработчиков корпоративных программных продуктов. В основе системы Maven лежит принцип *соглашения по конфигурации*, который упрощает стандартные варианты ее применения. Однако конфигурация

Maven не всегда обеспечивает гибкость. Впрочем, негибкость иногда оказывается преимуществом. Поскольку очень сложно изменить стандартную структуру проекта Maven и процесс сборки, большинство корпоративных Java-проектов получаются очень похожими. Новые программисты легко осваивают настройку сборки проекта.

На рис. 2.2 показан типичный пример структуры проекта Maven.

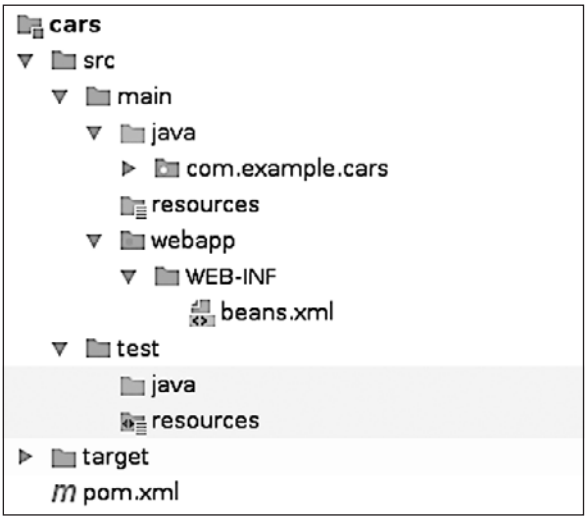

Рис. 2.2

Эта структура знакома большинству разработчиков корпоративных Java-проектов. Здесь приведен пример веб-приложения, упакованного в виде WAR-файла.

Одним из недостатков Apache Maven является несколько непрозрачный способ определения нужных расширений сборки и их зависимостей. Применение стандартного соглашения без явного указания версий для расширений, таких как *Maven Compiler Plugin*, может вызвать нежелательные изменения используемых версий. Это нарушает принцип повторяемости сборки. Из-за этого в проектах, требующих воспроизводимости, часто явно указывают и переопределяют версии зависимостей расширений в POM. В этом случае проекты каждый раз собираются с использованием одних и тех же версий, даже если меняются версии расширений по умолчанию.

Еще один распространенный способ определения точных версий расширений — применение Super POM. Объектные модели проекта могут наследоваться от родительских проектов и тем самым сокращать шаблонные определения расширений. Программисты могут задействовать эффективные представления POM, отражающие результирующую объектную модель, с учетом стандартной конфигурации и возможного наследования.

Типичная проблема с POM в Maven заключается в том, что в корпоративных проектах очень часто злоупотребляют XML-определениями. В них преждевременно вводят расширения или настройки, которые и так содержатся в стандартной конфигурации.

В следующем фрагменте кода показаны минимальные требования POM для проекта Java EE 8:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example.cars</groupId>
    <artifactId>car-manufacture</artifactId>
    <version>1.0.1</version>
    <packaging>war</packaging>

    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>8.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <finalName>car-manufacture</finalName>
    </build>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <failOnMissingWebXml>false</failOnMissingWebXml>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

</project>
```

Приложение для производства автомобилей *car manufacture* собрано в WAR-артефакт. `FinalName` переопределяет неявное имя WAR-файла, и получается файл `car-manufacturing.war`.

API Java EE 8 — единственная зависимость, необходимая в готовом коде простого корпоративного решения. Подробнее о зависимостях и их влиянии на проект мы поговорим в главе 4.

Тег `properties` устраняет необходимость явной настройки расширений сборки. Он используется для настройки параметров расширений Maven в каждой конфигурации. Указание этих параметров позволяет перенастроить расширение без явного объявления полных определений.

Указанные свойства позволяют построить проект с применением Java SE 8, причем считается, что все исходные файлы созданы в кодировке UTF-8. WAR-файлу не требуется передавать дескриптор развертывания `web.xml`, именно поэтому мы даем Maven указание не прерывать сборку при отсутствии дескриптора. Прежде для Servlet API требовались дескрипторы развертывания, чтобы настроить и сопоставить сервлеты приложения. После появления Servlet API версии 3 дескрипторы `web.xml` больше не нужны — сервлеты настраиваются с помощью аннотаций.

Процесс сборки в Maven состоит из нескольких этапов: компиляции, тестирования и упаковки. То, какие операции выполняются, зависит от выбранного этапа. Например, если выбрать этап упаковки, то будут скомпилированы исходные коды `main` и `test`, проведены контрольные тесты, после чего все классы и ресурсы будут упакованы в артефакт.

Команды сборки Maven запускаются через IDE или в виде командной строки `mvn`. Например, команда `mvn package` запускает этап упаковки, и на выходе создается упакованный артефакт. Подробнее об этапах и функциях системы Apache Maven вы можете узнать из ее официальной документации.

Gradle

На момент написания этой книги система Gradle применялась в корпоративных Java-проектах реже, чем Apache Maven. Возможно, это связано с тем, что разработчики корпоративных приложений зачастую незнакомы с динамическими JVM-языками, такими как Groovy, который используется в Gradle в качестве языка сценариев сборки. Однако для того, чтобы создавать файлы сборки Gradle, глубокое знание Groovy не требуется.

У Gradle множество преимуществ, и в первую очередь гибкость. Для определения и возможной настройки сборки проекта к услугам разработчиков все возможности языка программирования.

У Gradle в фоновом режиме работает демон, который после первой сборки запускается повторно, чтобы ускорить последующие сборки. Он также отслеживает входы и выходы сборки, проверяя, были ли внесены изменения со времени последнего ее выполнения. Это позволяет системе кэшировать операции, сокращая время сборки.

Однако такая оптимизация может и не потребоваться. Это определяется сложностью проекта и используемыми зависимостями. Подробнее о влиянии зависимостей проекта и приложениях с нулевой зависимостью читайте в главе 4.

На рис. 2.3 показана структура сборки проекта в Gradle. Как видите, есть большое сходство с проектами Maven, с той разницей, что готовые двоичные файлы после сборки по умолчанию помещаются в каталог `build`.

Обычно проекты Gradle включают в себя сценарий оболочки для тех операционных сред, в которых не установлена система Gradle.

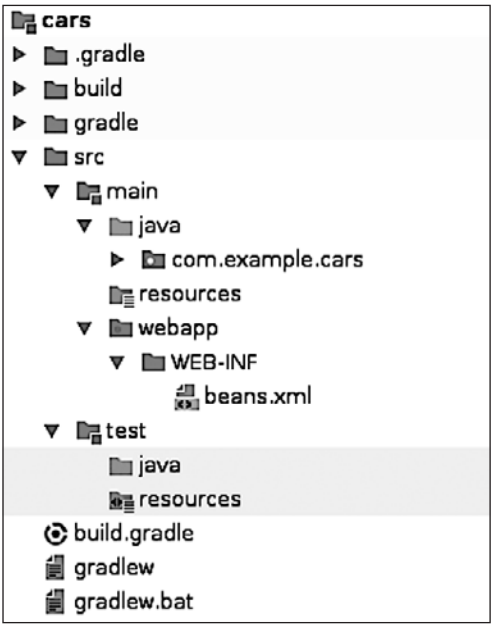

Рис. 2.3

Следующий код представляет собой пример файла `build.script`:

```
plugins {
    id 'war'
}

repositories {
    mavenCentral()
}

dependencies {
    providedCompile 'javax:javaee-api:8.0'
}
```

Задачи сборки в Gradle запускаются из командной строки с использованием команды `gradle` или посредством предоставляемых сценариев оболочки. Например, команда `gradle build` является аналогом `mvn package` и запускает компиляцию исходного кода, выполнение тестов и сборку артефакта.

Благодаря наличию полноценного языка программирования, определяющего файлы сборки, у Gradle есть ряд преимуществ. В случае обработки сценариев сборки в виде кода программистам удобнее применять принципы чистого кода, если определения становятся слишком сложными. Например, можно представить сложные операции сборки в виде нескольких методов с легко читаемым кодом.

Однако эти возможности порождают опасность чрезмерного технического усложнения сборки. Как уже отмечалось, негибкость Apache Maven иногда может быть преимуществом: возможность легко настраивать сценарии сборки позволяет создавать определения, присущие только данному проекту. Здесь, в отличие от Maven, перегруженные специфическими деталями сборки могут стать препятствием для программистов, незнакомых с проектом.

Опыт показывает, что подавляющее большинство сборок корпоративных проектов очень похожи. Поэтому возникает вопрос: так ли уж нужна гибкость Gradle? Для проектов, не предъявляющих особых требований к сборке, в отличие от, например, разработки продукта, вполне достаточно такой системы сборки, как Maven.

В дальнейшем, когда в качестве примера потребуется система сборки, мы будем использовать Maven. Однако все приводимые здесь примеры кода одинаково хорошо подходят и для Gradle.

Структурирование для современных клиентских технологий

После того как мы осветили современные системы сборки для корпоративных проектов, посмотрим, как интегрировать в серверную часть клиентские технологии.

Это делается довольно просто. Клиентская часть веб-приложений в большинстве случаев представляет собой создаваемые на сервере HTML-страницы со сценариями на JSP или JSF. HTML-страницы генерируются по мере необходимости, то есть по запросу клиента, и предоставляются ему. Для этого на сервере и располагаются JSP- или JSF-страницы. Все корпоративные приложения должны поставляться и развертываться как единый артефакт.

Введение в JavaScript-фреймворки

Благодаря новым технологиям разработки клиентской части, в основном сложным фреймворкам JavaScript и особенно одностраничным приложениям, эта ситуация сильно изменилась. Фреймворки для разработки веб-интерфейсов становятся все более клиентоориентированными и включают в себя гораздо больше бизнес-логики, чем прежде. На стороне сервера это означает, что взаимодействие между серверной и клиентской частью переходит от мелких методов к более крупным, описывающим бизнес-сценарии использования.

Таким образом, чем более клиентоориентированными и функциональными становятся фреймворки JavaScript, тем дальше взаимодействие между клиентской и серверной частью приложения уходит от тесно связанных запросов и ответов и приближается к стилю API, обычно в формате JSON через HTTP. Это также означает, что серверная сторона все меньше зависит от клиентской. Например,

при передаче данных только через RESTful-подобные API в формате JSON собственные и мобильные клиенты, такие как смартфоны, могут применять тот же API, что и у клиентской части.

Мы наблюдаем эту тенденцию во многих корпоративных проектах. Тем не менее еще неизвестно, что лучше, переносить все больше логики на сторону клиента или создавать гибридные решения, часть которых реализована на стороне сервера, а часть — на стороне клиента. Не вдаваясь в детали, рассмотрим несколько ключевых моментов.

Подготовка данных и содержимого быстрее выполняется на стороне сервера. Там больше возможностей и ресурсов, чем на стороне клиента. На сервере также доступны такие функции, как кэширование, и есть возможность увидеть ситуацию в целом.

Сложные клиентские технологии часто включают в себя логику навигации, которая использует так называемые шебанг-страницы. Примером URL шебанг-страницы является `/car-manufacturing/#!/Cars/1234`. Такие страницы, например `car 1234`, не размещаются на сервере, а генерируются на стороне клиента и существуют только там. URL-адрес такой подстраницы определяется после символа решетки и не учитывается при запросе ресурсов через HTTP. Это означает, что клиент запрашивает начальную входную страницу, которая затем реализует всю навигационную логику, включая генерацию подстраниц. Таким образом, с одной стороны, значительно уменьшается количество запросов, но с другой — сервер не обеспечивает подготовку и предварительную визуализацию содержимого — все происходит на стороне клиента. Именно по этой причине некоторые крупные компании, такие как Twitter, поначалу поддерживавшие такой подход, впоследствии отказались от него. В частности, просмотр этих страниц на мобильных устройствах сопряжен с определенными трудностями. Из-за ожидаемо медленных мобильных соединений и меньшей вычислительной мощности визуализация и выполнение сложной клиентской логики на этих устройствах занимают больше времени, чем отображение предварительно визуализированных HTML-страниц.

По сравнению с языками высокого уровня со статическим контролем типов, такими как Java, проблема клиентских интерфейсов на JavaScript заключается в том, что в языках с динамическим контролем типов выше вероятность пропустить ошибки программирования, которые иначе были бы обнаружены на стадии компиляции. По этой причине появляются более сложные клиентские технологии, такие как TypeScript, со статическими типами и языковые конструкции более высокого уровня, обработка которых передается обратно в JavaScript.

Структура современных клиентских приложений

Впрочем, независимо от того, какая именно технология выбрана для разработки клиентской части, сейчас клиентская часть корпоративных проектов гораздо сложнее, чем в прошлом. Это связано с новыми задачами организации ежедневного рабочего процесса. Обычно рабочие циклы клиентского интерфейса

и серверной части немного различаются. При этом одни программисты занимаются преимущественно серверной частью, а другие — клиентской. Даже если вся команда состоит исключительно из разработчиков полного цикла, со временем происходит разделение обязанностей.

Поэтому в зависимости от используемой технологии имеет смысл выделить клиентскую часть в особый проект. Как уже отмечалось, если часть программного обеспечения поставляется отдельно или его жизненный цикл отличается от жизненного цикла остальной системы, имеет смысл создать для него особый модуль проекта.

Если клиентская часть может быть развернута независимо от серверной части, за исключением использования HTTP, то структура проекта довольно проста. Проект можно собрать и развертывать на веб-сервере отдельно и задействовать одну или несколько серверных частей со стороны клиента. Такой проект состоит только из статических ресурсов, например файлов HTML, JavaScript или CSS, которые передаются клиенту и выполняются на его стороне. Здесь нет жестких технических зависимостей от применяемых серверных технологий, за исключением HTTP API.

Этот момент, очевидно, должен быть тщательно оговорен во время разработки, а также описан в документации серверной части. Как правило, серверная часть определяет HTTP-ресурсы, предоставляющие требуемое содержимое в формате JSON, которое при необходимости может быть отфильтровано в соответствии с параметрами запроса. Причиной столь широкого распространения формата JSON является то, что клиентский код JavaScript может использовать ответ сервера непосредственно в виде объектов JavaScript, без дополнительных преобразований.

Если клиентский интерфейс развертывается вместе с серверной частью как единый артефакт, то структура проекта требует большей координации. Артефакт содержит оба уровня технологии, они компилируются и собираются в пакеты одновременно в процессе сборки. При разработке такое объединение не является обязательным, если цикл разработки клиентской части отличается от разработки серверной. Программист, который занимается клиентской частью, едва ли захочет каждый раз пересобирать вместе с ней серверную часть. То же самое касается и серверной части, разработчики которой не захотят ждать окончания компиляции и упаковки потенциально медленного JavaScript.

В этих случаях имеет смысл разделить проект на несколько модулей, каждый из которых собирался бы отдельно. Хорошо зарекомендовал себя принцип упаковки клиентской части как отдельного модуля и представления его в виде зависимости для серверного модуля, который затем пакуется отдельно. Таким образом, клиентский модуль может быть собран отдельно, а программист серверной части может пересобирать серверную часть, используя последнюю версию клиентского интерфейса. Благодаря этому сокращается время сборки обеих частей.

Для того чтобы это можно было реализовать, в Servlet API предусмотрены статические ресурсы, не только упакованные в архив, но и содержащиеся в JAR-файлах. Ресурсы, находящиеся в разделе `META-INF/resources` JAR-файла,

который хранится в WAR-файле, также поставляются в контейнере Servlet. Проект клиентской части содержит все необходимые интерфейсные технологии, фреймворки и инструменты и собирается в отдельном JAR-файле. Это позволяет программистам отделять клиентскую часть от серверной, чтобы адаптироваться к их разным жизненным циклам.

Последующий материал будет посвящен серверным технологиям и бизнес-сценариям, доступ к которым можно получить посредством межкомпьютерного взаимодействия, например с помощью веб-сервисов.

Структура кода корпоративного проекта

Мы рассмотрели варианты структуры корпоративного проекта, теперь подробнее изучим конкретную структуру проекта. Предполагая, что мы моделируем корпоративную систему небольшого размера с незначительной функциональностью, преобразуем задачи проекта в кодовые структуры.

Мы уже знаем, что такое вертикальные и горизонтальные структуры модулей. Именно это нужно в первую очередь изучить при структурировании проекта.

Ситуация в корпоративных проектах

Типичный корпоративный проект обычно состоит из трех технически обоснованных слоев — уровней представления, бизнес-логики и данных. Это означает, что проект имеет горизонтальную структуру в виде трех подмодулей, или пакетов.

Идея состоит в том, чтобы разделить по уровням данные, бизнес-логику и представления. Таким образом, функциональность нижнего уровня не может зависеть от более высокого — только наоборот. На уровне бизнес-логики не может быть задействована функциональность уровня представления, а уровень представления может использовать функционал бизнес-логики. То же самое верно для уровня данных, который не может зависеть от уровня бизнес-логики.

У каждого технически обоснованного уровня или модуля есть свои внутренние зависимости, которые нельзя задействовать извне. Например, использовать базу данных может только уровень данных, прямые вызовы с уровня бизнес-логики недопустимы.

Еще одна причина разделения на уровни заключается в возможности передавать детали реализации, не влияя на другие уровни. Если будет принято решение изменить технологию базы данных, то теоретически это не повлияет на два остальных уровня, поскольку эти детали инкапсулированы на уровне данных. Аналогично изменение технологии представления никак не повлияет на остальные уровни. В сущности, можно создать даже несколько уровней представления, использующих на уровне бизнес-логики одни и те же компоненты, — в случае если эти уровни представлены в виде отдельных модулей.

Мы были свидетелями горячих дискуссий между высококвалифицированными архитекторами о необходимости организации и разделения обязанностей в соответствии с техническими задачами. Однако у этого подхода есть ряд недостатков.

Структурирование по горизонтали и по вертикали

Чистый код — это такой код, который должны понимать люди, а не машины. То же самое касается области разработки и разделения ответственности. Мы хотим построить структуру, по которой инженеры легко разобрались бы в том, что собой представляет проект.

Проблема структурирования по техническим признакам на высоких уровнях абстракции заключается в том, что при этом назначение и область применения программного обеспечения искажаются и скрываются на более низких уровнях абстракции. Когда человек, незнакомый с проектом, смотрит на структуру кода, первое, что он видит, — это три технических уровня (иногда их названия и количество могут быть иными). С одной стороны, это будет выглядеть знакомо, но с другой — ничего не скажет о фактической области применения программного продукта.

Инженеры-программисты стремятся разобраться в области применения модулей, а не в технических уровнях. Например, в функционале учетных записей они найдут только то, что связано с областью учетных записей, а не все подряд классы для доступа к базе данных. Более того, программисту, скорее всего, нужны не все классы доступа к базе данных, а только тот единственный класс, который обрабатывает эту логику в своей предметной области.

То же самое касается изменений, вносимых в систему. Изменения функциональности, скорее всего, влияют на все технические уровни одной или нескольких областей бизнес-логики, но вряд ли сразу на все классы одного технического уровня. Например, изменение одного поля в учетной записи пользователя повлияет на модель последнего, доступ к базе данных, бизнес-сценарии и даже логику представления, но не обязательно на все остальные классы моделей.

Для того чтобы лучше объяснить, какие характеристики больше всего интересуют программистов, позвольте привести еще один пример. Представьте, что все члены семьи хранят свою одежду в одном большом шкафу. Они могли бы разместить в одном ящике все брюки, в другом — все носки, а в третьем — все рубашки. Но мало кому захочется, одеваясь, перебирать все брюки, чтобы найти свои. Каждого интересует только его одежда, будь то брюки, рубашки, носки или что-то еще. Поэтому целесообразно сначала разделить шкаф на несколько зон, по одной на каждого члена семьи, а затем структурировать каждую зону по *техническим признакам* одежды, в идеале следуя аналогичной структуре. То же самое верно и в отношении программного обеспечения.

Структура, продиктованная бизнес-логикой

Роберт Мартин, он же дядюшка Боб, однажды описал *«кричащую» архитектуру*, которая должна сообщить инженеру прежде всего о том, что представляет собой корпоративный проект в целом. Чтобы, глядя на чертежи зданий, видя структуру и детализированный интерьер, сразу можно было сказать: *вот это дом, это библиотека, а это — железнодорожная станция*. То же самое справедливо и для программных систем. Взглянув на структуру проекта, программист должен сразу понять: *это система учета, это система инвентаризации книжного магазина, а это — система управления заказами*. Так ли это в большинстве существующих проектов? Или же, глядя на самый высокий уровень модулей и пакетов, мы можем лишь сказать: *это приложение Spring, у этой системы есть уровень представления, бизнес-логики и данных, а в этой системе используется кэш Hazelcast*?

Техническая реализация, безусловно, важна для нас, программистов. Однако первое, на что мы обращаем внимание, — бизнес-логика. Соответственно, эти аспекты должны отражаться в структуре проекта и модулей.

Но самое главное, это означает, что предметная область будет отражена в структуре приложения. Уже имена пакетов верхнего уровня иерархии должны давать представление о том, для чего предназначено это программное обеспечение. Поэтому разделение на уровни должно определяться прежде всего задачами бизнеса и только потом — деталями реализации.

Чертежи зданий тоже создаются по такому принципу: сначала нужно показать, что представляет собой здание, как разделены комнаты, где находятся двери и окна. И только потом могут быть указаны используемые материалы, сорта кирпича и тип бетона.

В качестве обзора микросервисов рассмотрим следующее: проектирование вертикальных модулей позволяет команде разработчиков легко представить приложение как систему из нескольких приложений. Исследуя зависимости между модулями — например, посредством анализа статического кода, — можно наметить точки интеграции между системами. Эти точки проявятся в какой-то форме коммуникации между приложениями. Теоретически можно создать один модуль, добавив к нему минимальные *связи*, и упаковать его в виде отдельного самодостаточного приложения.

Уточним терминологию: *модулями* мы сейчас называем бизнес-модули, реализованные в виде пакетов и подпакетов Java, а не модули проекта. Поэтому термином *«модуль»* описывается скорее концепция, чем строгая техническая реализация.

Рациональное проектирование модулей

А теперь перейдем к практике: как построить структуру модулей разумного размера?

Поскольку на первом месте у нас требования бизнеса, лучше всего начать с обзора всех задач и сценариев использования приложения. Это можно сделать во

время мозгового штурма, в идеале при участии специалистов по бизнесу, если это не было предпринято раньше. Каковы задачи приложения? Какие бизнес-сценарии следует реализовать? Какой подходящий функционал можно применить? Ответив на эти вопросы, вы составите представление о том, какие модули, скорее всего, будут присутствовать в проекте, не концентрируясь на внешних системах, деталях реализации или фреймворках.

На этом этапе мы уже учитываем зависимости между бизнес-задачами. Такие зависимости — полезные индикаторы того, следует модули разделить или же объединить, особенно если обнаружены круговые зависимости. Построение обзорных диаграмм, начиная с самого высокого уровня и постепенно спускаясь, за несколько итераций позволяет составить более четкое представление о том, какую бизнес-логику должно реализовывать приложение. Вообще говоря, идентифицированные модули должны соответствовать бизнес-логике, созданной экспертами предметной области.

Например, приложение для интернет-магазина может состоять из модулей, отвечающих за *пользователей*, *рекомендации*, *статьи*, *платежи* и *доставку*. Их основная структура может выглядеть так, как показано на рис. 2.4.

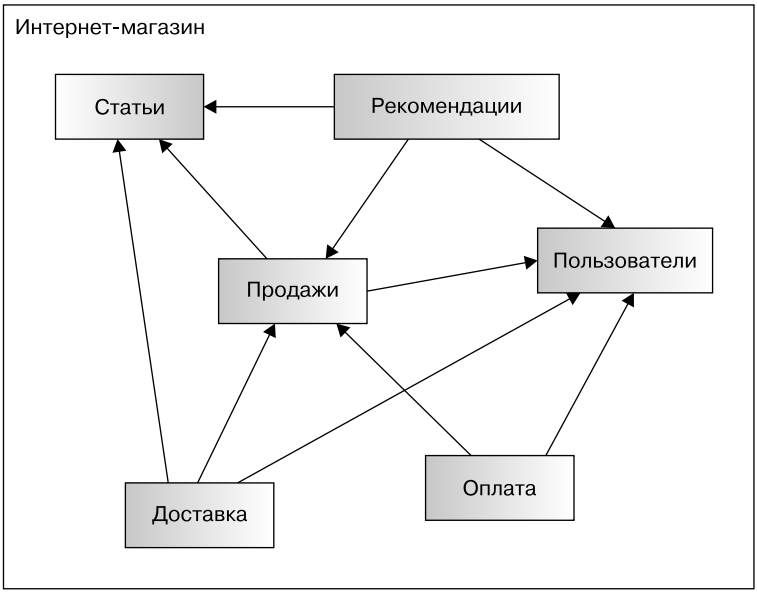

Рис. 2.4

Идентифицированные модули представляют собой базовые пакеты Java создаваемого приложения.

Имеет смысл потратить время на эту разработку. Однако, как обычно, любая *определенная* структура или реализация, независимо от уровня — кода или модулей, — должна иметь возможность изменяться. Могут появиться новые требования или сформироваться лучшее понимание уже существующих, когда программисты

тщательнее изучат предметную область. Итеративное перепроектирование, на каком бы уровне оно ни делалось, всегда улучшает качество системы.

В главе 8 будут показаны аналогичные причины и методологии проектирования систем с распределенными приложениями. В частности, мы рассмотрим принципы проблемно-ориентированного проектирования взаимосвязанных контекстов и карты контекстов.

Реализация пакетных структур

Предположим, что базовая структура Java-пакетов создана. Как теперь реализовать внутреннюю структуру пакетов? Другими словами, какие подпакеты использовать?

Содержимое пакетов

Для начала рассмотрим содержимое модуля с вертикальной структурой. Поскольку его модель соответствует бизнес-логике, то в состав модуля будет входить все необходимое для реализации определенного функционала.

Прежде всего в модуле будут технические точки входа для таких сценариев, как конечные точки HTTP, контроллеры представления фреймворков или конечные точки JMS. В этих классах и методах обычно применяются такие принципы Java EE, как инверсия управления, чтобы при обращении к приложению можно было осуществлять вызов прямо из контейнера.

Следующей не менее важной задачей является создание функциональности, реализующей различные сценарии использования. Обычно они отличаются от технических конечных точек тем, что не содержат коммуникационной логики. Бизнес-сценарии являются точкой входа в логику предметной области. Они реализуются как управляемые компоненты, обычно как Stateless Sessions Beans — другими словами, EJB или управляемые компоненты CDI.

Эти компоненты очерчивают пределы, в которых реализуется бизнес-логика. В тех случаях, когда логика сценария состоит всего из нескольких шагов, она вполне может быть ограничена лишь бизнес-методом или приватными методами в определении класса. В этом случае никакие другие делегаты не нужны. В подавляющем большинстве бизнес-сценариев за логику отвечают соответствующие службы, и именно эти делегаты выполняют более мелкие операции. В зависимости от предметной области это означает реализацию детальной бизнес-логики или доступ к внешним системам, таким как базы данных. В соответствии с языком проблемно-ориентированного проектирования эти классы включают в себя службы, сценарии транзакций, фабрики и хранилища данных.

Следующий тип объектов — это все классы, которые обычно принято считать *моделями* содержимого, такого как сущности, объекты-значения и объекты переноса данных. Эти классы описывают сущности из предметной области, но также могут и должны реализовывать бизнес-логику. Примерами являются сущностные объекты, которыми оперирует база данных, другие POJO-объекты и перечисления.

В некоторых случаях в состав пакета может входить также сквозная функциональность, такая как методы-перехватчики бизнес-логики или технических функций. Теперь остается разместить и упорядочить все эти компоненты в рамках модуля.

Горизонтальное структурирование пакетов

Если бы перед нами стояла задача структурировать содержимое модуля, то сначала мы попытались бы, вероятно, спроектировать внутреннюю структуру пакета путем технического разделения на уровни. Сперва разделить по коммерческим задачам, а затем по техническим функциям — по крайней мере, это звучит разумно.

Например, в пакете `users` это означало бы предусмотреть такие подпакеты, как `controller`, `business` (или `core`), `model`, `data` и `client`. Придерживаясь этого подхода, далее разделяем функционал внутри пакета `users` по техническим категориям. Действуя последовательно, мы добьемся того, что все остальные модули и пакеты в проекте будут иметь аналогичную структуру в зависимости от их содержимого. Эта идея похожа на реализацию трехуровневой архитектуры, но только внутри модулей, соответствующих предметной области.

Один из подпакетов, например `controller`, будет считаться технической точкой входа. Этот пакет будет содержать коммуникационные конечные точки, инициирующие логику сценариев использования, и служить точкой входа извне приложения. На рис. 2.5 приведена структура горизонтально упорядоченного пакета `users`.

На рис. 2.6 эта структура реализована в виде Java-пакетов.

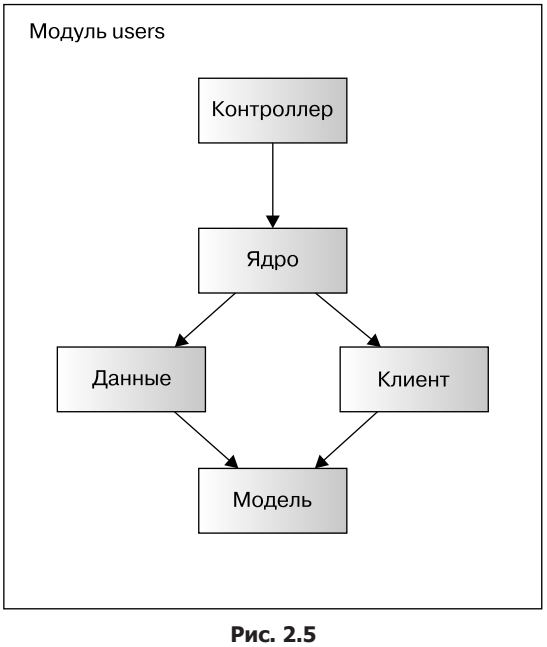

Рис. 2.5

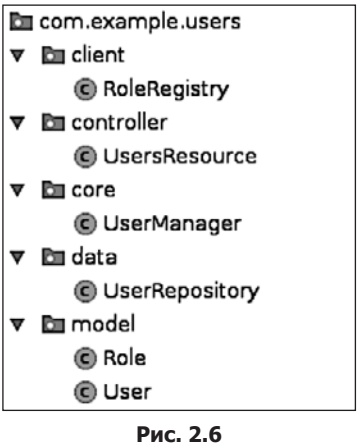

Рис. 2.6

Плоская структура модулей

Есть более простой и прямолинейный способ структурировать содержимое модуля — разместить все связанные классы в модульном пакете в виде плоской (горизонтальной) иерархии. В частности, для пакета `users` это означает включить в него все классы, относящиеся к пользователям, в том числе точки доступа для пользовательских сценариев, параметры доступа к базе данных пользователей, потенциальный функционал для внешних систем и классы пользовательских сущностей.

В зависимости от сложности модулей таким способом можно получить либо ясную и простую структуру, либо такую, которая со временем превратится в хаотичную. В частности, сущности, объекты-значения и объекты переноса данных могут расшириться до нескольких классов. Если все эти классы находятся в одном пакете, то его структура становится запутанной и непонятной. Тем не менее имеет смысл начать с такой структуры и при необходимости позже ее реорганизовать.

На рис. 2.7 показана структура рассмотренного в примере пакета `users`.

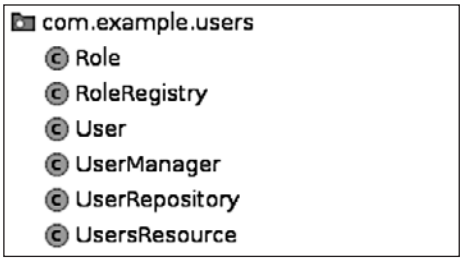

Рис. 2.7

Преимущество такого подхода заключается в том, что он хорошо поддерживается языком Java. По умолчанию классы и методы Java имеют видимость только в пределах пакета. С учетом этого обстоятельства размещение всех классов в одном пакете улучшает инкапсуляцию и видимость. Компоненты, к которым желательно обеспечить доступ извне пакета, становятся общедоступными. Все классы и методы, которые должны быть доступны только внутри пакета, имеют такую видимость по умолчанию. Таким образом, в пакете инкапсулируются все внутренние задачи.

Подход «Сущность — управление — граница»

Для того чтобы можно было справиться с таким большим количеством классов в пакете модуля, предусмотрен другой подход, аналогичный техническому разделению на слои, но более логичный и подразумевающий использование меньшего количества пакетов. Смысл его состоит в том, чтобы создать структуру модулей, соответствующую контурам (границам) бизнес-сценариев, которые затем станут компонентами бизнес-логики и классами сущностей.

При этом основное внимание уделяется структурированию пакетов модулей в соответствии с их задачами, но с меньшим, по сравнению с горизонтальным структурированием, количеством технических деталей на верхнем уровне пакета. В *контуре* содержатся инициаторы бизнес-сценариев и классы-разграничители, доступные извне системы. Эти классы обычно представляют собой конечные точки HTTP, управляемые сообщениями объекты, контроллеры клиентских интерфейсов и обычные серверные компоненты Java. Они реализуют бизнес-сценарии и, возможно, делегируют какие-то операции подчиненным классам из дополнительных пакетов *управления*. Пакет *сущности* содержит все имена модуля, сущности предметной области и объекты переноса данных.

Айвар Джекобсон (Ivar Jacobson) сформулировал термин *«сущность — управление — граница» (Entity Control Boundary, ECB)*, описывающий показанный на рис. 2.8 способ организации модулей.

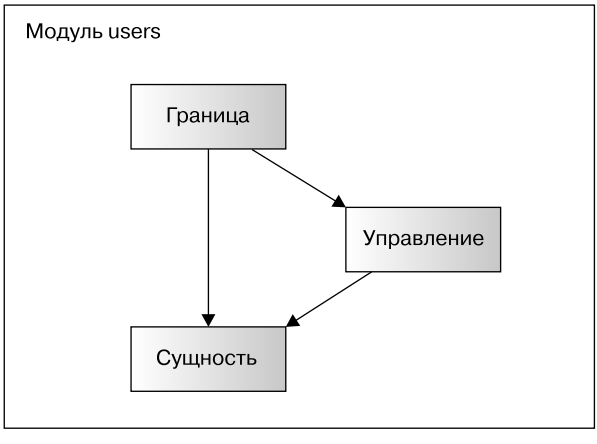

Рис. 2.8

Пакеты

Рассмотрим подробнее пограничные пакеты, соответствующие контурам. Суть в том, чтобы инициировать в таких пакетах все бизнес-сценарии, вызываемые из клиентского интерфейса или извне системы. Эти пакеты принимают обращения для создания, обновления или удаления пользователей, им же принадлежат классы верхнего уровня. В зависимости от сложности бизнес-сценариев контур либо полностью обрабатывает логику, либо делегирует ее элементу управления, прежде чем это станет слишком сложным.

В корпоративном приложении Java классы пограничного пакета реализованы в виде управляемых объектов. Как уже отмечалось, в данном случае обычно применяется EJB.

Если логика в контуре становится слишком сложной и трудноуправляемой в рамках одного класса, то ее передают задействованным в контуре делегатам.

Эти делегаты, или *элементы управления*, размещаются в пакете управления. Они обычно реализуют более детальные бизнес-сценарии, обеспечивают доступ к базе данных или внешним системам, действуя в рамках технических транзакций, которые инициируются контуром.

В такой структуре повышаются согласованность и возможность повторного использования, а также соблюдается принцип разделения обязанностей. После того как введены эти уровни абстракции, структура бизнес-сценариев становится более читаемой. Сначала можно рассматривать контур как точку входа в бизнес-сценарий, а затем по очереди выполнять все делегированные операции.

В языке проблемно-ориентированного проектирования пакет управления включает в себя службы, сценарии транзакций, фабрики и хранилища данных. Однако наличие пакета управления для реализации бизнес-сценариев не является обязательным.

Основа нашего продукта — сущности и объекты-значения. В совокупности с объектами передачи данных они образуют модель модулей предметной области — объектов, которыми обычно оперирует бизнес-сценарий. Они объединены в пакет сущностей в соответствии с шаблоном ECB.

А как быть с компонентами, связанными с презентацией, и сквозными задачами, такими как перехватчики или фреймворки со связующей логикой? К счастью, в современных проектах Java EE все требуемые связующие функции спрятаны в пределах фреймворка, как мы увидим в главе 3. Всего несколько необходимых вещей, таких как самонастройка JAX-RS с классом активатора приложения, помещаются в корневой пакет проекта или в специальный пакет `platform`. То же самое касается и сквозных задач, таких как технически обусловленные методы-перехватчики, которые не привязаны к определенному модулю, а относятся к приложению в целом. Обычно таких классов не очень много, имеет смысл выделить их в особый пакет. Опасность добавления пакета `platform` заключается в том, что программистам, естественно, захочется добавить туда и другие компоненты. Однако он предназначен лишь для нескольких классов платформы — все остальное должно находиться в соответствующих пакетах бизнес-логики.

На рис. 2.9 приведен пример модуля `users` с использованием шаблона ECB.

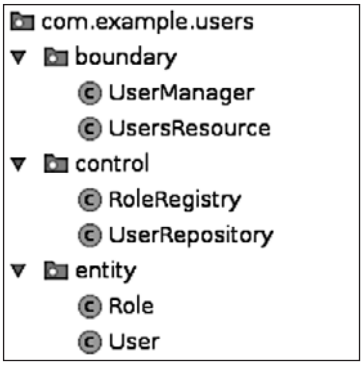

Рис. 2.9

Доступ к пакетам

В шаблоне ECB далеко не каждое обращение из каждого пакета имеет смысл и, соответственно, считается допустимым. В общем случае логический поток начинается с контура и заканчивается на пакете управления или пакете сущностей. Таким образом, у пограничного пакета есть зависимости от пакета управления (если он существует) и пакета сущностей. Использование пограничных пакетов других модулей не допускается и не имеет смысла, так как подобный пакет представляет собой отдельный бизнес-сценарий. Доступ к другому пограничному пакету означал бы обращение к тому, что должно быть отдельным, автономным бизнес-сценарием. Поэтому пограничные пакеты могут обращаться только *вниз* по иерархии к элементам управления.

Однако иногда зависимости и вызовы от пограничных пакетов к элементам управления других модулей допустимы и имеют смысл. Программистам нужно следить за тем, чтобы при обращении к компонентам из других модулей правильно выбиралась область транзакций. При доступе к элементам управления из других модулей также нужно проследить, чтобы объекты, с которыми они работают или которые возвращают, принадлежали соответствующим «чужим» модулям. Так бывает в нестандартных бизнес-сценариях, и это не вызывает проблем, если позаботиться о разделении обязанностей и правильном использовании элементов управления и сущностей.

Элементы управления одного модуля могут обращаться к элементам управления других модулей, а также к их собственным и внешним сущностям. Как и в случае с пограничными пакетами, нет смысла в том, чтобы элемент управления вызывал функции контура. Это означало бы появление нового бизнес-сценария верхнего уровня в рамках уже существующего сценария.

Сущности могут зависеть только от других сущностей. Иногда приходится импортировать элементы управления, например, если есть приемник сущностей JPA или преобразователи типа JSON-B, реализующие сложную логику. Такие технически обоснованные случаи, когда следует разрешить импортировать классы, являются исключением из общего правила. В идеале *поддерживающие сущности* компоненты, такие как приемники или преобразователи сущностей, должны находиться непосредственно в пакете сущностей. Из-за наличия других зависимостей и использования делегирования данное правило не всегда выполняется, но это не должно приводить к реализации чрезмерно сложных технических решений.

Все это подводит нас еще к одной важной теме.

Не перегружайте архитектуру

Какой бы архитектурный шаблон вы ни выбрали, основным приоритетом приложения должна оставаться его бизнес-логика. Это справедливо как при поиске приемлемых модулей, соответствующих предметной области, так и при структурировании пакетов внутри модуля, чтобы программисты могли работать с ним с минимальными усилиями.

Важно: программисты должны работать над проектом, не задействуя слишком сложных или чрезмерно перегруженных структур и архитектур. Нам встречалось слишком много примеров того, как намеренно использовались уровни, где выполнялись чисто технические задачи, или слишком строгие шаблоны — лишь бы сделать все по книге или учесть заданные ограничения. Но эти ограничения часто ничем не обоснованы и не помогают достичь какой-либо важной цели. Необходимо рационально все пересмотреть и отделить то, что действительно необходимо, от того, что лишь раздувает процесс разработки. Поинтересуйтесь на досуге, что означает термин «карго-культ» в программировании, — вы услышите интересную реальную историю о том, что бывает, если следовать правилам и ритуалам, не задумываясь об их смысле.

Поэтому не стоит чрезмерно усложнять или перегружать архитектуру. Если существует простой и понятный способ достичь того, что требуется в настоящее время, — просто воспользуйтесь им. Это относится не только к преждевременному рефакторингу, но и к проектированию архитектуры. Если при объединении нескольких классов в один пакет с понятным названием будет решена поставленная задача и обеспечено создание понятной документации, почему бы не ограничиться этим? Если пограничный класс для бизнес-сценария сам выполняет всю простую логику, зачем создавать пустые делегаты?

Если архитектурный шаблон не всегда нужен, то необходимо искать компромисс между последовательностью и простотой. Если все пакеты, модули и проекты соответствуют одним и тем же шаблонам и имеют одинаковую структуру, то у программистов быстро складывается представление об архитектуре. Однако в главе 8 мы увидим, что при ближайшем рассмотрении согласованность — это цель, которой вряд ли удастся достичь в рамках всей организации или даже отдельных проектов. Во многих случаях выгоднее разработать что-то более простое и, следовательно, более гибкое, чем сохранить единообразие.

То же самое верно и для чрезмерной инкапсуляции с использованием технических уровней. Действительно, именно модули и классы должны инкапсулировать детали реализации, обеспечивая ясные и понятные интерфейсы. Однако эти задачи могут и должны решаться с помощью единых, в идеале самодостаточных пакетов и классов. Упаковывать в модуль всю ключевую функциональность неправильно, поскольку, с технической точки зрения, в таком случае мы открываем доступ к деталям в остальной части модуля, например, выдаем, какую базу данных мы используем или через какой клиент общаемся с внешней системой. Построение структуры системы в первую очередь в соответствии с задачами предметной области позволяет инкапсулировать функционал в единых точках ответственности, прозрачных для остальных модулей и приложения в целом.

Во избежание ошибок при выборе способа упаковки следует воспользоваться самым простым и прозрачным вариантом — выполнить статический анализ кода. Пакеты, импортированные в классы, и сами пакеты можно сканировать и анализировать с целью обнаружения и предотвращения нежелательных зависимостей. Такое определение степени безопасности, похожее на использование тестовых сценариев, позволяет избежать случайных ошибок. Статический анализ кода

обычно выполняется как дополнительная процедура в процессе сборки на сервере непрерывной интеграции, поскольку сборка занимает определенное время. Мы подробно рассмотрим эту тему в главе 6.

Резюме

Главным приоритетом при разработке корпоративного программного обеспечения должна быть реализация бизнес-логики. Это приводит к построению приложений и технологий на базе бизнес-сценариев, а не технологических решений. В конечном счете именно бизнес-сценарии принесут прибыль.

Насколько возможно, корпоративные приложения должны разрабатываться в рамках одного проекта сборки для каждого артефакта в системе контроля версий. Разделение проекта на несколько независимых модулей сборки, которые в итоге сводятся к одному артефакту, не приносит особой пользы. На верхних уровнях структуры проекта рекомендуется располагать программные модули вертикально, а не горизонтально. Это означает строить структуру приложения в соответствии с бизнесом, а не с техническими задачами. С первого взгляда на структуру проекта программист должен понимать, к какой предметной области относится приложение и какие задачи оно выполняет.

В самом простом случае отдельный модуль приложения может представлять собой единый Java-пакет. Это удобно, если количество классов в модуле невелико. Для более сложных модулей имеет смысл создать дополнительный иерархический уровень, используя такой шаблон, как ECB.

Инженерам-программистам следует помнить, что не нужно чрезмерно усложнять архитектуру программного продукта. Хорошо продуманная и задокументированная структура, безусловно, способствует созданию высококачественного программного обеспечения. Тем не менее всегда есть золотая середина между разумной архитектурой и чрезмерным техническим усложнением.

Рассмотрев основную структуру корпоративных проектов и способы разработки модулей, спустимся на один уровень и разберем, как создаются модули проекта. В следующей главе поговорим о том, что нужно для реализации корпоративных приложений на базе Java EE.

3 Внедрение современных приложений Java Enterprise

Теперь, когда мы знаем, какие компоненты содержатся в проектах и модулях, по какому принципу строятся и как разрабатываются модули и пакеты приемлемых размеров, перейдем от теории к практике и поговорим о Java EE. Разумеется, имеет смысл сначала обсудить задачи бизнеса и следовать рекомендациям проблемно-ориентированного проектирования, чтобы определить область применения и модули для всех задач приложения.

Посмотрим, как можно реализовать намеченные бизнес-модули и сценарии использования.

В этой главе будут рассмотрены следующие темы.

- Как очертить границы бизнес-сценариев приложения.
- Что такое бизнес-компоненты ядра Java EE.
- Шаблоны проектирования и проблемно-ориентированное проектирование в Java EE.
- Коммуникация в приложении.
- Как интегрировать долговременное хранение данных.
- Технические сквозные задачи и асинхронное поведение.
- Концепции и принципы Java EE.
- Как получить обслуживаемый код.

Границы бизнес-сценариев

Построение пакетов в соответствии с задачами предметной области приводит нас к архитектурной структуре, отражающей реальные коммерческие задачи, а не технические подробности.

Бизнес-сценарии описывают всю логику, необходимую для выполнения задач предприятия, используя для этого содержимое модуля. Они играют роль от-

правной точки в работе приложения. Бизнес-сценарии реализуются на границах системы. Корпоративные системы предоставляют коммуникационные интерфейсы для связи с внешним миром в основном через веб-сервисы и клиентские веб-интерфейсы, которые вызывают соответствующий бизнес-функционал.

Работу над новым проектом имеет смысл начать с логики предметной области, не заботясь о границах (контуре) системы и других деталях технической реализации. Это означает, что сначала разрабатывается все содержимое предметной области, проектируются типы, зависимости и задачи, а потом создаются их прототипы в виде кода. Как мы увидим в этой главе, фактическая логика предметной области реализуется в основном в виде обычного кода Java. Первоначальная модель может быть самодостаточной и тестироваться исключительно с использованием тестов на уровне кода. После того как будет построена довольно полная модель предметной области, можно переключиться на оставшиеся технические проблемы, выходящие за пределы модуля бизнес-логики, такие как доступ к базам данных и внешним системам, а также конечные точки системы.

В приложении Java EE контур реализуется с применением управляемых объектов — *Enterprise JavaBeans (EJB)* или *Contexts and Dependency Injection for Java (CDI)*. В разделе «EJB и CDI: общее и различия» мы рассмотрим, чем эти технологии отличаются одна от другой и в чем их назначение.

Для сложных бизнес-сценариев иногда создают *делегатов*, реализованных в виде управляемых объектов CDI или EJB в зависимости от требований. Эти делегаты находятся в пакете управления. Сущности реализуются в виде POJO, иногда с описаниями, чтобы интегрировать такие технические функции, как отображение базы данных или сериализация.

Бизнес-компоненты ядра в современной Java EE

Простая Java, CDI и EJB — это базовые бизнес-компоненты в современном приложении на Java EE. Почему они называются базовыми бизнес-компонентами? Как уже отмечалось, в первую очередь мы обращаем внимание на реальные коммерческие задачи. Существуют компоненты и функции, нацеленные на решение основных бизнес-задач, тогда как другие просто *поддерживают*, делают их доступными или реализуют иные технические требования.

Java EE поставляется в комплекте с различными API, которые поддерживают десятки технических требований. Большинство из них технически обоснованы. Однако самым большим преимуществом платформы Java EE является то, что чистая бизнес-логика на Java может быть реализована с минимальным воздействием технологических деталей на код. Обычно для этого достаточно CDI и EJB. Остальные API, необходимые по техническим соображениям, такие как JPA, JAX-RS, JSON-P и многие другие, вводятся по мере необходимости.

Управляемые объекты независимо от типа — CDI или EJB — реализованы в виде аннотированных классов Java, без каких-либо технических суперклассов или интерфейсов. Раньше это называлось представлением без интерфейса. В настоящее время этот вариант применяется по умолчанию. Расширение классов усложняет представление предметной области, не говоря уже о других недостатках, которые проявляются при тестировании. Современный фреймворк интегрируется максимально простым и незаметным способом.

EJB и CDI: общее и различия

Зададимся вопросом: EJB или CDI? Какие управляющие объекты применять и когда?

Вообще говоря, у EJB больше функциональных возможностей, готовых к использованию. Управляемые объекты CDI представляют собой несколько упрощенную альтернативу. Каковы основные различия между этими технологиями и как они влияют на работу программиста?

Первое различие — это области видимости. Объекты сессии EJB либо не сохраняют состояние, то есть остаются активными в течение всего клиентского запроса, либо сохраняют состояние, то есть остаются активными в течение времени жизни HTTP-сессии клиента, либо являются синглтонами. Управляемые объекты CDI имеют дополнительные возможности, такие как добавление нестандартных областей действия, а область видимости, выбираемая по умолчанию, является зависимой — она активна в зависимости от времени жизни ее точки внедрения. Подробнее об этом читайте в подразделе «Области видимости».

Другое различие между управляемыми объектами EJB и CDI заключается в том, что EJB-объекты неявно решают определенные сквозные задачи, такие как мониторинг, транзакции, обработка исключений и управление параллелизмом для синглтонов. Например, при вызове бизнес-метода EJB неявно запускает техническую транзакцию, которая остается активной в течение всего времени выполнения метода и объединяет источники данных или внешние системы.

Кроме того, EJB без сохранения состояния после использования объединяются. Это означает, что после вызова бизнес-метода для объекта без сохранения состояния сущность этого объекта может и будет применяться повторно из контейнера. Благодаря этому EJB-объекты немного эффективнее, чем CDI, которые заново создаются каждый раз, когда их область видимости требует этого.

На практике технические различия не очень сильно влияют на работу программиста. За исключением применения разных аннотаций, обе технологии могут использоваться в одном и том же стиле. Java EE движется к возможности предоставлять более открытый выбор из этих двух вариантов. Например, начиная с версии Java EE 8, можно обрабатывать асинхронные события исключительно с помощью CDI, а не только привлекая EJB-объекты.

Однако интеграция функциональности от CDI — одна из самых серьезных возможностей API Java EE. Уже только внедрение зависимостей, CDI-генераторы и события являются эффективными средствами для решения различных задач.

Главная, наиболее широко применяемая функция CDI — это внедрение зависимостей с помощью аннотации `@Inject`. Оно реализовано таким образом, что программисту не нужно задумываться, какая технология Java EE управляет объектами, — она *просто работает*. Вы можете смешивать и комбинировать CDI-объекты и EJB с любыми областями видимости — фреймворк сам позаботится о том, какие объекты создаются или используются в данной области видимости. Это обеспечивает гибкость, например, когда объекты с более короткой областью видимости внедряются в объекты с более длинной областью видимости — допустим, когда объект с областью видимости в пределах сессии внедряется в синглтон.

Данная функция поддерживает бизнес-задачи таким образом, что контуры и элементы управления могут просто внедрять нужные зависимости, не занимаясь их созданием или управлением ими.

В следующем коде показано, как контур, реализованный в виде объекта сессии без сохранения состояния, внедряет необходимые элементы управления:

```
import javax.ejb.Stateless;
import javax.inject.Inject;

@Stateless
public class CarManufacturer {

    @Inject
    CarFactory carFactory;

    @Inject
    CarStorage carStorage;

    public Car manufactureCar (Specification spec) {
        Car car = carFactory.createCar (spec);
        carStorage.store (car);
        return car;
    }
}
```

Класс `CarManufacturer` представляет собой EJB-объект без сохранения состояния. Внедренные компоненты `CarFactory` и `CarStorage` реализованы в виде CDI-объектов с зависимыми областями видимости, которые будут созданы и внедрены в EJB-объект. Платформа Java EE упрощает устранение зависимостей, позволяя с помощью аннотации `@Inject` вставлять любые объекты, необходимые для проекта. Но так было не всегда — прежде для внедрения EJB-объектов задействовалась аннотация `@EJB`. Аннотация `@Inject` упрощает использование объектов в Java EE.

Внимательные читатели могли заметить, что внедрение через поле с областями видимости осуществляется внутри пакета Java. Внедрение через поле меньше влияет на содержимое класса, поскольку позволяет не обращаться к его конструктору. Видимость внутри пакета дает программистам возможность создавать и внедрять зависимости в тестовой области видимости. Подробнее эта технология и ее альтернативы будут рассмотрены в главе 7.

Генераторы CDI

Генераторы CDI — это еще одна функция Java EE, которая особенно полезна для реализации всевозможных фабрик. Генераторы, в основном реализуемые как методы, создают объект, который можно внедрять в другие управляемые объекты. Это отделяет логику создания и конфигурации от использования. Генераторы полезны в случаях, когда нужно вводить дополнительные типы, помимо тех, которые внедряются управляемыми объектами.

Далее показано определение метода-генератора CDI:

```
import javax.enterprise.inject.Produces;

public class CarFactoryProducer {

    @Produces
    public CarFactory exposeCarFactory() {
        CarFactory factory = new BMWCarFactory();
        // дополнительная логика
        return factory;
    }
}
```

Открытый тип `CarFactory` можно ввести просто с помощью внедрения аннотацией `@Inject`, как было показано ранее в примере с `CarManufacturer`. Каждый раз, когда нужна сущность `CarFactory`, CDI-объект вызывает метод `exposeCarFactory()` и вставляет возвращаемый объект в точку внедрения.

Этих методов достаточно для большинства случаев использования основной бизнес-логики.

Генерация событий предметной области

Для тех случаев, когда необходимо еще больше отделить бизнес-логику от управляемых объектов, в CDI предусмотрена функция обработки событий. Управляемые объекты могут активизировать объекты событий, которые затем действуют как полезная нагрузка и обрабатываются наблюдателями событий. Активизируя и обрабатывая события CDI, мы отделяем основную бизнес-логику от побочных аспектов обработки событий. Этот прием применяется, в частности, в тех случаях, когда в бизнес-логике уже продумана концепция событий.

По умолчанию события CDI обрабатываются синхронно, прерывая выполнение в том месте, где они были активизированы. Они также могут обрабатываться асинхронно или в определенных точках жизненного цикла технической транзакции.

На примере следующего кода показано, как определять и активизировать события CDI в бизнес-сценарии:

```
import javax.enterprise.event.Event;

@Stateless
public class CarManufacturer {

    @Inject
    CarFactory carFactory;

    @Inject
    Event<CarCreated> carCreated;

    public Car manufactureCar(Specification spec) {
        Car car = carFactory.createCar(spec);
        carCreated.fire(new CarCreated(spec));
        return car;
    }
}
```

Событие `CarCreated` является неизменяемым и содержит относящуюся к событию предметной области информацию, такую как спецификация автомобиля. Само событие обрабатывается в классе `CreatedCarListener`, который находится в следующем пакете управления:

```
import javax.enterprise.event.Observes;

public class CreatedCarListener {

    public void onCarCreated(@Observes CarCreated event) {
        Specification spec = event.getSpecification();
        // обработка события
    }
}
```

Таким образом, обработчик события отделен от основной бизнес-логики. Контейнер CDI сам обеспечит подключение функций обработки событий и синхронный вызов метода `onCarCreated()`.

Подробнее об активизации и обработке событий асинхронно или в определенных точках жизненного цикла транзакции читайте в разделе «Последовательность выполнения».

События CDI — это способ отделить определение событий предметной области от их обработки. Тогда можно изменить или усилить логику обработчика событий, не затрагивая компонент `manufactureCar`.

Области видимости

Область видимости управляемых объектов очень важна в тех случаях, когда состояние сохраняется в приложении дольше чем на протяжении одного запроса.

Если весь бизнес-процесс можно реализовать без сохранения состояния, просто выполняя некоторую логику и затем отбрасывая все состояния, то определить область видимости очень легко. В большинстве случаев для этого достаточно объектов сессии без сохранения состояния и зависимых от области видимости объектов CDI.

Области видимости синглтона EJB и приложения CDI также довольно часто используются. Отдельные сущности управляемых объектов предоставляют простой способ хранения или кэширования информации с длительным временем жизни. Наряду со сложной технологией кэширования, синглтон, содержащий простые наборы или отображения с контролем параллелизма, по-прежнему остается простейшим способом разработки кратковременных хранилищ данных, ориентированных на решение конкретных задач. Синглтоны также обеспечивают единую точку запуска функций, доступ к которым по определенным причинам должен быть ограничен.

И последняя область видимости управляемых объектов EJB и CDI — это область сессии, привязанная к HTTP-сессии клиента. Объекты из этой области остаются активными и могут быть повторно использованы вместе со всеми своими состояниями, пока активна сессия пользователя. Однако при хранении данных сессии в объектах с сохранением состояния возникает проблема, когда клиенты заново подключаются к тому же серверу приложений.

Это, безусловно, возможно, но не позволяет создавать приложения без сохранения состояния, которыми было бы легко управлять. Если приложение становится недоступным, то все временные данные сессии теряются. В современных корпоративных приложениях в целях оптимизации состояние обычно хранится в базе данных или кэшах. Поэтому объекты с видимостью в пределах сессии применяются все реже.

Управляемые объекты CDI имеют больше готовых областей видимости, а именно область видимости в пределах обмена данными и зависимую область видимости (используется по умолчанию). Существует также возможность создания нестандартных областей видимости для особых требований. Однако опыт показывает, что встроенных областей видимости достаточно для большинства корпоративных приложений. Подробная информация о том, как расширить платформу и разработать дополнительные области видимости, содержится в спецификации CDI.

Как мы уже видели, с помощью основных компонентов Java EE можно достичь очень многого. Прежде чем приступать к изучению технологий интеграции, таких как HTTP-коммуникации и доступ к базе данных, более подробно рассмотрим основные шаблоны, применяемые при проектировании предметной области.

Шаблоны проектирования в Java EE

О шаблонах проектирования написано много. Одним из самых ярких и часто упоминаемых источников является книга «Паттерны проектирования»[1], написанная *«бандой четырех»* (Gang of Four, GoF). В ней описываются наиболее распространенные задачи разработки программного обеспечения и шаблоны реализации, с помощью которых эти задачи решаются.

Структура и обоснование использования конкретных шаблонов по-прежнему актуальны, но их фактическая реализация во многом изменилась, особенно в области корпоративного программного обеспечения. Помимо распространенных шаблонов проектирования, которые применимы для всех видов приложений, появилось множество других, связанных исключительно с корпоративными программными продуктами. В частности, ранее существовало множество корпоративных шаблонов для J2EE. Но мы живем во времена Java EE 8, а не J2EE, и теперь появились более простые способы реализации различных шаблонов для решения конкретных задач.

Обзор шаблонов проектирования

Шаблоны проектирования, описанные в книге GoF, делятся на три категории: порождающие, структурные и поведенческие. Каждый шаблон описывает типичную задачу, решаемую программным обеспечением, и показывает способ ее решения. Шаблоны представляют собой схемы реализации, не зависящие от конкретной технологии. Вот почему любой из этих шаблонов может быть реализован в любой среде. В современном мире Java SE 8 и EE 8 больше языковых возможностей, чем прежде. Я хочу показать некоторые шаблоны проектирования, предложенные «бандой четырех», обосновать их применение и объяснить, как они могут быть реализованы в Java EE.

Синглтон

Объект-одиночка (singleton) — хорошо известный шаблон или, по мнению некоторых, антишаблон. Синглтоны имеют только одну сущность для каждого класса во всем приложении. Такой шаблон задействуют, когда нужны возможности сохранения состояний, а также координации всех действий в едином центре. Синглтоны, безусловно, имеют право на существование. Если нужно обеспечить надежный совместный доступ нескольких потребителей к определенному состоянию или создать единую точку входа, то самым простым решением такой задачи будет реализация синглтона.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2015.

Здесь следует учитывать несколько моментов. Наличие единой точки ответственности создает параллелизм, которым необходимо управлять. Поэтому синглтоны должны быть поточно-ориентированными. Другими словами, необходимо помнить, что синглтоны по своей природе не масштабируются, поскольку существует только один экземпляр класса. Чем большую синхронизацию мы вводим из-за структуры данных, тем хуже наш класс будет справляться с одновременным доступом. Однако в зависимости от сценария использования это может и не быть проблемой.

В книге GoF описывается статический синглтон, управляемый одноэлементным классом. В Java EE концепция синглтонов встроена непосредственно в EJB в виде сессионных синглтонов и в CDI — в виде области видимости приложения. Эти определения создают один управляемый объект, применяемый всеми клиентами.

Рассмотрим пример синглтона EJB:

```
import javax.ejb.Singleton;

@Singleton
public class CarStorage {

    private final Map<String, Car> cars = new HashMap<>();
    public void store(Car car) {
        cars.put(car.getId(), car);
    }
}
```

Есть важное различие между синглтонами, реализованными на основе управляемого объекта EJB или объекта области видимости приложения CDI.

По умолчанию параллелизмом синглтонов EJB управляет контейнер. Это гарантирует, что открытые бизнес-методы будут выполняться по одному по очереди. Поведение может быть изменено с помощью аннотации `@Lock`, которая объявляет методы как блокирующие запись или чтение, — другими словами, когда управляемые объекты действуют как блокираторы записи или чтения соответственно. Все бизнес-методы синглтона EJB неявно блокируют запись. Далее приведен пример использования EJB с контейнером, управляемым параллелизмом, и аннотацией блокировки:

```
import javax.ejb.Lock;
import javax.ejb.LockType;

@Singleton
public class CarStorage {

    private final Map<String, Car> cars = new HashMap<>();

    @Lock
    public void store(Car car) {
        cars.put(car.getId(), car);
    }

    @Lock(LockType.READ)
```

```
    public Car retrieve(String id) {
        return cars.get(id);
    }
}
```

Параллелизм также можно отключить, заменив его на параллелизм, управляемый объектом. Объект будет вызываться на конкурентной основе, а сама его реализация обеспечит безопасность потоков. Так, использование потокобезопасной структуры данных не требует, чтобы синглтон EJB управлял параллельным доступом. Бизнес-методы сущности EJB будут вызываться параллельно аналогично объектам CDI, принадлежащим области видимости приложения, как показано далее:

```
import javax.ejb.ConcurrencyManagement;
import javax.ejb.ConcurrencyManagementType;

@Singleton
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class CarStorage {

    private final Map<String, Car> cars = new ConcurrentHashMap<>();

    public void store(Car car) {
        cars.put(car.getId(), car);
    }

    public Car retrieve(String id) {
        return cars.get(id);
    }
}
```

CDI-объекты с областью видимости в пределах всего приложения не ограничивают параллельный доступ, и при их реализации всегда необходимо учитывать параллелизм.

Эти решения применяются в ситуациях, когда требуется один синглтон, например, для состояния, которое должно храниться в памяти и быть доступным в пределах всего приложения.

CDI-объекты, видимые в пределах всего приложения, и синглтоны EJB с управляемым параллелизмом и потокобезопасными структурами данных позволяют создать в памяти некластеризованный, доступный в пределах всего приложения кэш, который очень хорошо масштабируется. Если не требуются распределенные вычисления, то это простейшее, но элегантное решение.

Еще одним достоинством применения синглтонов EJB является возможность вызывать общий процесс при запуске приложения. С помощью аннотации `@Startup` создается готовый к запуску приложения компонент, вызывающий метод `@PostConstruct`. Процессы запуска могут быть определены для всех EJB, но, задействуя синглтоны, можно реализовать процессы, которые нужно настраивать только один раз.

Абстрактная фабрика

Шаблон абстрактной фабрики, описанный GoF, направлен на то, чтобы отделить создание объектов от их применения. Создание сложных объектов может требовать знаний об определенных предварительных условиях, деталях реализации или того, какой именно класс реализации будет задействован. Фабрики помогают создавать такие объекты, не обладая глубокими знаниями об их внутренней структуре. Позже в этой главе мы обсудим фабрики в проблемно-ориентированном проектировании, которые тесно связаны с этим шаблоном. Причины использования фабрик здесь те же: абстрактные фабрики, как правило, имеют несколько реализаций абстрактного типа, где сама фабрика также является абстрактным типом. Пользователи функционала разрабатывают интерфейсы, тогда как конкретная фабрика будет создавать и возвращать конкретные сущности.

Например, можно создать абстрактную фабрику `GermanCarFactory` с такими конкретными реализациями, как `BMWFactory` и `PorscheFactory`. Обе фабрики автомобилей будут выполнять реализацию `GermanCar`, только в одном случае это будет `BMWCar`, а в другом — `PorscheCar`. Клиенту, которому нужен просто немецкий автомобиль, не придется задумываться о том, какой из конкретных классов реализации выберет фабрика.

В мире Java EE уже есть мощный функционал, который, в сущности, является фабричной структурой, — это CDI. CDI предоставляет множество функций для создания и внедрения сущностей определенных типов. Причины их применения и результаты одинаковы, но детали реализации различаются. На практике существует множество способов реализации абстрактной фабрики в зависимости от сценария использования. Рассмотрим некоторые из них.

Управляемый объект может внедрять сущности, которые могут быть конкретными, абстрактными и даже параметризованными типами. Если нам нужна только одна сущность в данном объекте, то внедряем непосредственно `GermanCar`:

```
@Stateless
public class CarEnthusiast {

    @Inject
    GermanCar car;

    …
}
```

Если будет введено несколько реализаций типа `GermanCar`, это приведет к исключению разрешения зависимостей, поскольку контейнер не может знать, какой именно автомобиль нужно внедрить. Для того чтобы решить эту проблему, можно ввести квалификаторы, которые явно запрашивают конкретный тип. Можно воспользоваться готовым классификатором `@Named`, определив для него строковые значения, однако это небезопасно с точки зрения типов. CDI позволяет создавать собственные типобезопасные классификаторы в соответствии с бизнес-сценарием:

```
@BMW
public class BMWCar implements GermanCar {
    …
}

@Porsche
public class PorscheCar implements GermanCar {
    …
}
```

Квалификаторы представляют собой пользовательские аннотации, сохраняемые при выполнении приложения, которые, в свою очередь, аннотируются с помощью `@Qualifier` и обычно `@Documented`:

```
import javax.inject.Qualifier;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Qualifier
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface BMW {
}
```

Квалификаторы описаны в точке внедрения. Они определяют внедряемый тип и отделяют внедрение от фактически используемого типа:

```
@Stateless
public class CarEnthusiast {

    @Inject
    @BMW
    GermanCar car;

    …
}
```

При получении сущности `CarEnthusiast` теперь будет создан и внедрен объект `BMWCar` с зависимой областью видимости, поскольку этот тип соответствует точке внедрения.

Теперь можно даже определить подтип автомобиля BMW, который использовался бы без изменения точки внедрения. Это делается путем создания *специализации* типа `BMWCar` с другой реализацией. Тип `ElectricBMWCar` является подклассом для класса `BMWCar` и описан с помощью аннотации `@Specializes`:

```
import javax.enterprise.inject.Specializes;

@Specializes
public class ElectricBMWCar extends BMWCar {
    …
}
```

Специализированные объекты наследуют типы и квалификаторы родительского типа и могут прозрачно использоваться вместо него. В представленном

примере внедрение `GermanCar` с классификатором `@BMW` предоставляет сущность `ElectricBMWCar`.

Однако для того, чтобы быть ближе к описанному в книге шаблону проектирования, мы могли бы также определить тип автомобильной фабрики, который при необходимости позволял бы создавать сразу несколько автомобилей:

```
public interface GermanCarManufacturer {
    GermanCar manufactureCar();
}
```

Эта автомобильная фабрика может быть реализована с различными спецификациями:

```
@BMW
public class BMWCarManufacturer implements GermanCarManufacturer {

    @Override
    public GermanCar manufactureCar() {
        return new BMWCar();
    }
}

@Porsche
public class PorscheCarManufacturer implements GermanCarManufacturer {

    @Override
    public GermanCar manufactureCar() {
        return new PorscheCar();
    }
}
```

Благодаря этому теперь клиент при создании новых немецких автомобилей сможет внедрять и использовать производителя напрямую:

```
@Stateless
public class CarEnthusiast {

    @Inject
    @BMW
    GermanCarManufacturer carManufacturer;

    // создать немецкую машину
}
```

Если типы внедрений явно определены и квалифицированы, как приведенные в примере два немецких автомобиля, это обеспечивает бóльшую гибкость при реализации.

Метод фабрики

Для того чтобы понять, что представляет собой метод фабрики, рассмотрим другой шаблон, с похожим назначением, но иначе реализованный. Методы фабрики определяют фабрики, реализованные как методы конкретных типов. Не существует

единого класса, ответственного за создание определенных сущностей, — создание сущностей является обязанностью метода фабрики, описанного как часть класса предметной области.

В качестве примера рассмотрим автомобиль, в котором все поездки записываются в водительский журнал. Было бы весьма разумно включить в тип автомобиля метод `createDriverLog()`, который возвращал бы объект типа «водительский журнал», поскольку вся его логика может быть реализована в пределах данного класса. Такие решения удобно воплощать на чистом Java, без использования каких-либо фреймворков или аннотаций:

```
public class Car {

    …

    public LogBook createDriverLog() {
        // создание бортового журнала
    }
}
```

Как мы увидим далее, при проблемно-ориентированном проектировании фабрики не делятся на абстрактные фабрики и методы фабрики. Они больше ориентированы на требования предметной области. Иногда имеет смысл инкапсулировать фабрики как методы вместе с другими обязанностями класса. В других случаях, когда логика создания объектов является особой точкой ответственности, лучше выделить ее в отдельный класс. Вообще говоря, желательно включать логику создания объектов в типы предметной области, поскольку тогда можно использовать другие функциональные возможности и свойства этого класса предметной области.

Рассмотрим генераторы CDI-объектов. Генераторы — это методы или поля, которые динамически применяются для поиска и внедрения сущностей определенных типов. У нас есть полная свобода выбирать значения, содержащиеся в этом поле или методе. Мы также можем задавать квалификаторы, чтобы исключить конфликт с генераторами других типов, которые, возможно, появятся впоследствии. Управляемые объекты, которые определяют метод генератора, могут содержать также дополнительные свойства, используемые генератором:

```
import javax.enterprise.inject.Produces;

public class BMWCarManufacturer {

    …

    @Produces
    @BMW
    public GermanCar manufactureCar() {
        // использовать свойства
        …
    }
}
```

Это соответствует идее методов фабрики, реализованных в виде генераторов CDI.

Какова же область видимости генерируемых сущностей? Как и другие управляемые объекты CDI, генераторы по умолчанию зависят от области видимости. Последняя определяет жизненный цикл управляемых объектов и способы их внедрения. Это влияет на частоту вызова метода генерации. В области видимости, определяемой по умолчанию, метод вызывается один раз для каждой внедряемой сущности, когда создается вызываемый управляемый объект. Каждый раз, когда вводится объект, внедряющий сгенерированную сущность, вызывается метод генерации. Даже если у этого объекта длительный срок службы, метод генерации вызывается однократно.

Далее в этой главе мы рассмотрим более сложные способы использования генераторов CDI.

Накопитель объектов

Шаблон проектирования «Накопитель объектов» был разработан для оптимизации производительности. Накопители объектов создаются для того, чтобы избежать необходимости постоянно создавать новые сущности объектов и зависимостей, вместо этого сохраняя их в накопителе в течение более длительного времени. Требуемая сущность извлекается из накопителя и освобождается после использования.

Данная концепция уже встроена в контейнеры Java EE в разных формах. Как отмечалось ранее, объекты без сохранения состояния сессии помещаются в накопитель, именно это объясняет их высокую эффективность. Однако программистам следует знать, что эти сущности используются многократно и не должны сохранять состояние после того, как были задействованы. Накопитель этих сущностей хранится в контейнере.

Другим примером является накопитель соединений с базой данных. Соединения с базой данных устанавливаются долго, поэтому имеет смысл сохранить некоторые из них для использования в дальнейшем. В зависимости от реализации постоянного хранилища данных эти соединения повторно применяются при каждом следующем запросе.

В корпоративных приложениях существуют также накопители потоков. В серверной среде Java клиентский запрос обычно приводит к созданию потока Java, который обрабатывает его логику. После обработки запроса потоки могут использоваться повторно. Конфигурация накопителя потоков, а также наличие разных потоков — важный аспект дальнейшей оптимизации производительности. Мы рассмотрим этот вопрос в главе 9.

Обычно программисты не создают накопители объектов самостоятельно. Как правило, такой шаблон для сущностей, потоков и баз данных входит в состав контейнера, и разработчики приложений неявно применяют эти готовые функции.

Декоратор

Еще один известный шаблон проектирования — «Декоратор». Он позволяет добавлять поведение объекта, не затрагивая другие объекты этого класса. Очень часто такое поведение создается сочетанием нескольких подтипов.

При рассмотрении декоратора можно провести аналогию с едой. У каждого из нас свои вкусовые предпочтения. Возьмем, к примеру, кофе. Мы можем выпить обычный черный кофе, или кофе с молоком, или с сахаром, или с молоком и сахаром, или даже с сиропом. И это еще не учитывая различные способы приготовления кофе.

Далее показана реализация шаблона «Декоратор» на чистом Java.

Мы указываем тип `Coffee`, который может быть декорирован с использованием подтипа `CoffeeGarnish`:

```
public interface Coffee {

    double getCaffeine();
    double getCalories();
}

public class CoffeeGarnish implements Coffee {

    private final Coffee coffee;

    protected CoffeeGarnish(Coffee coffee) {
        this.coffee = coffee;
    }

    @Override
    public double getCaffeine() {
        return coffee.getCaffeine();
    }

    @Override
    public double getCalories() {
        return coffee.getCalories();
    }
}
```

Стандартная реализация `CoffeeGarnish` просто делегирует полномочия родительскому классу `Coffee`. Возможны следующие варианты приготовления напитка:

```
public class BlackCoffee implements Coffee {

    @Override
    public double getCaffeine() {
        return 100.0;
    }

    @Override
```

```
    public double getCalories() {
        return 0;
    }
}
```

Помимо класса для обычного черного кофе, можно создать классы для кофе с добавками:

```
public class MilkCoffee extends CoffeeGarnish {

    protected MilkCoffee(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCalories() {
        return super.getCalories() + 20.0;
    }
}

public class SugarCoffee extends CoffeeGarnish {

    protected SugarCoffee(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCalories() {
        return super.getCalories() + 30.0;
    }
}

public class CreamCoffee extends CoffeeGarnish {

    protected CreamCoffee(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCalories() {
        return super.getCalories() + 100.0;
    }
}
```

Используя разные типы кофе, мы можем составить желаемый вариант со специфическим поведением:

```
Coffee coffee = new CreamCoffee(new SugarCoffee(new BlackCoffee()));
coffee.getCaffeine(); // 100.0
coffee.getCalories(); // 130.0
```

Примером шаблона «Декоратор» в JDK является класс `InputStream` с возможностью добавления определенного поведения для файлов, байтовых массивов и т. п.

В Java EE мы снова используем CDI, который поставляется с функциональностью декоратора. Декораторы добавляют управляемому объекту особенное

поведение. При активизации внедренного объекта вместо фактического объекта вызывается декоратор, он добавляет специальное поведение и делегирует управление сущности объекта. Исходный тип управляемого объекта становится так называемым делегатом декоратора:

```
public interface CoffeeMaker {
    void makeCoffee();
}

public class FilterCoffeeMaker implements CoffeeMaker {

    @Override
    public void makeCoffee() {
        // сварить кофе
    }
}
```

Типом делегата должен быть интерфейс. `CountingCoffeeMaker` декорирует существующие функции кофеварки (класс `CoffeeMaker`):

```
import javax.decorator.Decorator;
import javax.decorator.Delegate;
import javax.enterprise.inject.Any;

@Decorator
public class CountingCoffeeMaker implements CoffeeMaker {
    private static final int MAX_COFFEES = 3;
    private int count;

    @Inject
    @Any
    @Delegate
    CoffeeMaker coffeeMaker;

    @Override
    public void makeCoffee() {
        if (count >= MAX_COFFEES)
            throw new IllegalStateException("Больше кофе сварить нельзя.");
        count++;

        coffeeMaker.makeCoffee();
    }
}
```

Функциональность декоратора активизируется с помощью дескриптора `beans.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
        bean-discovery-mode="all">
    <decorators>
        <class>com.example.coffee.CountingCoffeeMaker</class>
    </decorators>
</beans>
```

После активизации декоратора внедренные сущности типа `CoffeeMaker` используют декорированные функции. Это происходит без изменения исходной реализации:

```
public class CoffeeConsumer {

    @Inject
    CoffeeMaker coffeeMaker;

    …
}
```

Управляемые объекты могут иметь несколько декораторов. При необходимости можно задать последовательность использования декораторов с помощью аннотации `@Priority`.

В зависимости от того, требуется ли добавить поведение существующим классам моделей предметной области или связанным с ними сервисам, этот шаблон создают либо на чистом Java, как описано ранее, либо с помощью декораторов CDI.

Фасад

Шаблон проектирования «Фасад» позволяет создать понятный и простой интерфейс для определенных функций. Безусловно, одними из важнейших принципов при написании кода являются инкапсуляция и абстракция. Фасады позволяют инкапсулировать сложный функционал или громоздкие устаревшие компоненты, скрывая их за более простыми интерфейсами. Таким образом, фасад является ярким примером абстракции.

Рассмотрим весьма сложную настройку для кафе. В нем есть кофемолки, кофемашины, весы и различные приспособления, которые необходимо правильно настроить:

```
public class BaristaCoffeeShop {

    private BeanStore beanStore;
    private Grinder grinder;
    private EspressoMachine espressoMachine;
    private Scale scale;
    private Thermometer thermometer;
    private Hygrometer hygrometer;

    public GroundBeans grindBeans(Beans beans, double weight) { ... }

    public Beans fetchBeans(BeanType type) { ... }

    public double getTemperature() { ... }

    public double getHumidity() { ... }

    public Coffee makeEspresso(GroundBeans beans, Settings settings) { ... }
}
}
```

Можно с уверенностью утверждать, что этот класс уже сейчас нуждается в рефакторинге. Однако устаревшие классы не так легко изменить. Поэтому мы опишем баристу, который играет роль фасада:

```
@Stateless
public class Barista {

    @Inject
    BaristaCoffeeShop coffeeShop;

    public Coffee makeCoffee() {
        // проверить температуру и влажность
        // посчитать количество кофе и проверить настройку кофемашины
        // насыпать и смолоть кофе
        // сварить эспрессо
    }
}
```

В Java EE самым ярким примером фасадов являются контуры, реализуемые с помощью EJB. Они служат фасадами для бизнес-сценариев, которые являются частью предметной области. Кроме того, фасады могут быть реализованы с использованием всех видов управляемых объектов.

Фасады делегируют функции и организуют выполнение сложной логики. Правильно подобранные абстракции улучшают структуру программного продукта — при разработке надо стремиться к этому.

Посредник

Шаблон проектирования «Посредник», вероятно, наиболее очевидный из тех, что входят в состав Java EE. Ссылки на внедренные объекты почти всегда содержат ссылку не на фактическую сущность, а на посредника. Посредники — это тонкие обертки для сущностей, в которые могут быть добавлены определенные функции. Клиент даже не замечает, что взаимодействует не с самим объектом, а с посредником.

Посредники обеспечивают сквозную функциональность, которая требуется в корпоративной среде: сюда относятся перехватчики, транзакции, протоколирование или мониторинг. Но в первую очередь они необходимы для внедрения зависимостей.

Разработчики приложений обычно не задействуют шаблон «Посредник» напрямую. Тем не менее рекомендуется разобраться в том, как именно он работает в целом и как используется на платформе Java EE в частности.

Наблюдатель

Шаблон проектирования «Наблюдатель» описывает, как объект управляет наблюдателями и уведомляет их в случае изменения состояния. Наблюдатели регистрируются для получения уведомлений по определенным темам и затем

получают эти уведомления. Уведомления могут рассылаться наблюдателям в синхронном или асинхронном режиме.

Как мы уже знаем, CDI включает в себя функциональные возможности обработки событий, реализуя таким образом шаблон «Наблюдатель». Программистам не нужно самостоятельно разрабатывать логику регистрации и уведомления, они просто создают слабую связь, используя аннотацию. Как показано в разделе «Бизнес-компоненты ядра в современной Java EE», тип `Event<T>` и аннотация `@Observes` объявляют публикацию и наблюдение за событиями. Асинхронные события CDI будут рассмотрены в разделе «Последовательность выполнения».

Стратегия

Шаблон проектирования «Стратегия» используется для динамического выбора алгоритма реализации — стратегии — во время выполнения приложения. Он применяется, например, для выбора из нескольких бизнес-алгоритмов в зависимости от обстоятельств.

Есть несколько возможностей применения шаблона «Стратегия», которые зависят от ситуации. Можно оформить различные реализации алгоритма в виде отдельных классов. В Java SE 8 встроена функциональность лямбда-методов и ссылок на методы, которые можно использовать как простую реализацию стратегии:

```
import java.util.function.Function;

public class Greeter {

    private Function<String, String> strategy;

    String greet(String name) {
        return strategy.apply(name) + ", меня зовут Дюк";
    }

    public static void main(String[] args) {
        Greeter greeter = new Greeter();

        Function<String, String> formalGreeting = name -> "Дорогой " + name;
        Function<String, String> informalGreeting = name -> "Привет, " + name;

        greeter.strategy = formalGreeting;
        String greeting = greeter.greet("Java");

        System.out.println(greeting);
    }
}
```

В этом примере показано, что функциональные интерфейсы могут использоваться для динамического определения стратегий, которые выбираются и применяются в процессе выполнения приложения.

В среде Java EE для этого можно задействовать также внедрение зависимостей CDI. Чтобы продемонстрировать, что CDI поддерживает любые типы Java,

мы воспользуемся тем же примером со стратегией, который был приведен для функционального интерфейса. Тип `Function` представляет здесь стратегию приветствия:

```
public class Greeter {

    @Inject
    Function<String, String> greetingStrategy;

    public String greet(String name) {
        return greetingStrategy.apply(name);
    }
}
```

Метод генератора CDI динамически выбирает стратегию приветствия:

```
public class GreetingStrategyExposer {

    private Function<String, String> formalGreeting = name -> "Дорогой " + name;
    private Function<String, String> informalGreeting = name -> "Привет, " + name;

    @Produces
    public Function<String, String> exposeStrategy() {
        // выбор стратегии
        …
        return strategy;
    }
}
```

Чтобы пример был полным, добавим конкретные классы для реализации алгоритмов. CDI позволяет внедрять все сущности определенного типа, которые можно выбирать динамически.

Тип `GreetingStrategy` определяется в зависимости от времени суток:

```
public interface GreetingStrategy {
    boolean isAppropriate(LocalTime localTime);
    String greet(String name);
}

public class MorningGreetingStrategy implements GreetingStrategy {
    @Override
    public boolean isAppropriate(LocalTime localTime) {
        …
    }

    @Override
    public String greet(String name) {
        return "Доброе утро, " + name;
    }
}

public class AfternoonGreetingStrategy implements GreetingStrategy { ... }
public class EveningGreetingStrategy implements GreetingStrategy { ... }
```

Генератор CDI может внедрять любые сущности `GreetingStrategy` и выбирать их на основе спецификации:

```
public class GreetingStrategySelector {

    @Inject
    @Any
    Instance<GreetingStrategy> strategies;

    @Produces
    public Function<String, String> exposeStrategy() {
        for (GreetingStrategy strategy : strategies) {
            if (strategy.isAppropriate(LocalTime.now()))
                return strategy::greet;
        }
        throw new IllegalStateException("Не удалось выбрать тип приветствия");
    }
}
```

Квалификатор `@Any` неявно существует в любом управляемом объекте. Точки внедрения с типом `Instance` и этот квалификатор внедряют все сущности, соответствующие указанному типу, в данном случае `GreetingStrategy`. Тип `Instance` позволяет динамически получать и квалифицировать сущности заданного типа. Он реализует итератор по всем подходящим типам.

Благодаря специальной логике мы выбираем подходящую стратегию, которая затем внедряется в приветствие.

В CDI есть несколько способов описания и выбора различных стратегий. В зависимости от ситуации внедрение зависимости может применяться для отделения логики выбора от использования.

Другие шаблоны

Помимо упомянутых шаблонов, реализованных с применением функциональных возможностей Java EE, есть и другие шаблоны проектирования, которые все еще реализуются на чистом Java, как описано в книге GoF. Представленный здесь список является далеко не полным, но включает в себя шаблоны проектирования, которые обычно используются в корпоративных проектах.

Некоторые шаблоны проектирования, такие как «Посредник», заложены в основу Java EE. Другим примером является «Медиатор», который инкапсулирует обмен данными между набором объектов. Например, для разработки слабосвязанной коммуникации не нужно реализовывать этот шаблон самостоятельно — достаточно воспользоваться функционалом API, внутри которого он реализован, в частности событиями CDI.

Есть много других шаблонов, которые не получили широкого применения в Java EE API, но могут быть реализованы на чистом Java. В зависимости от конкретного случая можно использовать CDI для создания и инстанцирования объектов. Примерами являются шаблоны «Прототип», «Строитель», «Адаптер»,

«Мост», «Композит», «Приспособленец», «Цепочка ответственности», «Состояние» и «Посетитель».

И снова, если рассматривать Enterprise API, мы обнаружим, что, например, шаблон «Строитель» активно задействуется в API JSON-P. Подробнее о работе с шаблонами читайте в книге «Паттерны проектирования», написанной «бандой четырех».

Проблемно-ориентированное проектирование

Итак, теперь мы знаем, как шаблоны проектирования, описанные в книге GoF, реализованы в Java EE. Кроме этого, прежде чем переходить к чисто техническим вопросам, я хочу перечислить некоторые основные шаблоны и концепции, которые применяются при описании предметной области. Более подробно эти шаблоны и концепции, поддерживающие разработку программных моделей, максимально точно описывающих реальные бизнес-сценарии, описаны в книге Эрика Эванса (Eric Evans) «Проблемно-ориентированное проектирование»[1]. В частности, в ней обращается особое внимание на то, как важно постоянно консультироваться с экспертами предметной области и говорить с ними на понятном, *общепринятом* языке, хорошо понимать и постепенно совершенствовать базовую модель этой области. Проблемно-ориентированное проектирование вводит в мир программного обеспечения ряд концепций, таких как репозитории, сервисы, фабрики и агрегаты.

Возникают вопросы: можно ли реализовать эти концепции в Java Enterprise и как это сделать? Проблемно-ориентированное проектирование всегда стремится реализовать важные детали приложения непосредственно в модели предметной области, а не только *снаружи*, как часть сценария сервиса или транзакции. Мы увидим, что такой подход хорошо работает в управляемых объектах EJB и CDI.

Сервисы

Язык проблемно-ориентированного проектирования определяет концепцию сервиса. Сервисы отвечают за организацию различных процессов бизнес-логики. Как правило, они служат точкой входа для бизнес-сценариев и создают объекты, принадлежащие модели предметной области, или управляют ими. Сервисы объединяют отдельные этапы бизнес-процессов.

Если сопоставить эту концепцию с идеей и содержанием пакетов ECB, то увидим, что она решает ту же задачу, что и контуры и элементы управления. Таким образом, в Java EE эти сервисы реализованы в виде управляемых объектов EJB и CDI. Сервисы, представляющие точку входа в сценарий, реализованы в виде контуров, тогда как сервисы, обеспечивающие дальнейшее выполнение бизнес-логики, доступ к базе данных и внешним системам, представляют собой элементы управления.

---

[1] *Эванс Э.* Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. — Киев: Вильямс, 2011.

Сущности

Проблемно-ориентированное проектирование также определяет так называемые *сущности*. Как видно из названия, сущность представляет собой некое понятие, относящееся к предметной области. Сущности являются идентифицируемыми частями концепций, которые составляют основу данной предметной области. Примеры сущностей — пользователи, статьи, автомобили. Важно, чтобы сущности предметной области можно было различать. Нужно понимать, какой из пользователей вызвал определенный бизнес-сценарий — Джон Доу или Джон Смит. В этом отличие сущностей от объектов-значений.

Сущности, а также другие объекты модели реализуются в виде простых классов Java. Чтобы функционировать в единственной предметной области, им не требуется поддержка фреймворков. В идеале сущности сами по себе уже инкапсулируют определенную бизнес-логику, которая является самодостаточной для данного типа сущностей. Это означает, что можно моделировать не только простые POJO и их свойства, но и методы чтения и записи, а также бизнес-методы, которые работают с этой сущностью. Интеграция бизнес-логики непосредственно в ядро бизнес-сущностей повышает согласованность, делает структуру более понятной и соответствует принципу единой ответственности.

Обычно сущности, как и другие модельные типы предметной области, сохраняются в базе данных. В Java EE поддерживается объектно-реляционное отображение данных с JPA, которое используется для сохранения и извлечения объектов и объектных иерархий. Аннотация JPA для объявления типов сущностей так и называется — `@Entity`. Далее в этой главе мы подробно рассмотрим, как JPA поддерживает сохранение модельных типов предметной области с минимальным изменением классов модели.

Объекты-значения

Если тип предметной области образует не идентифицируемые сущности, а лишь определенные *значения*, то он называется объектом-значением. Предпочтительно, чтобы объекты-значения были неизменяемыми. Они могут использоваться повторно, поскольку их содержимое постоянно. Хороший пример объектов-значений — перечисления Java. Любые объекты, в которых идентичность неважна, могут быть реализованы как объекты-значения. Например, для перечислений Java неважно, какая из сущностей `Status.ACCEPTED` будет возвращена — есть только одна сущность перечисления, которая и применяется в любых случаях. То же самое верно для многих других типов предметной области, таких как адреса. Пока значение адреса `42 Wallaby Way, Sydney` остается неизменным, не столь важно, к какой именно сущности адреса мы обращаемся.

В зависимости от того, является ли набор значений конечным, объекты-значения реализуются либо как перечисления, либо как объекты POJO, в идеале неизменяемые. Неизменяемость представляет собой концепцию объектов-значений и уменьшает вероятность ошибок. Если объект изменяемый, то при его исполь-

зовании в нескольких местах изменение может привести к незапланированным побочным эффектам.

Поскольку объекты-значения не идентифицируются напрямую, то нельзя сохранять их и управлять ими непосредственно в базе данных. Они, безусловно, могут сохраняться косвенно, как часть графа объектов, на который ссылается сущность или агрегат. JPA поддерживает управление сохранением объектов, которые не являются ни сущностями, ни агрегатами.

Агрегаты

Агрегаты представляют собой концепцию языка проблемно-ориентированного проектирования, которая иногда вводит программистов в заблуждение. Агрегаты — это сложные модели, состоящие из нескольких сущностей или объектов-значений, которые образуют единое целое. В целях согласованности этот конгломерат объектов должен быть доступен, и управлять им следует также как чем-то целым. Непосредственный доступ к методам хранящихся в нем отдельных объектов может привести к несогласованности и потенциальным ошибкам. Идея агрегата заключается в том, чтобы все операции выполнялись через корневой объект. Хорошим примером агрегата является автомобиль, состоящий из четырех колес, двигателя, кузова и т. п. Всякий раз, когда требуется выполнить какую-либо операцию с автомобилем, например завести двигатель, эта операция будет вызываться для всего автомобиля и может затрагивать сразу несколько объектов.

Агрегаты — это сущности, в которых определен корень иерархии объекта. Они выполнены как простые классы Java, в которых реализованы функции из предметной области и имеется ссылка на сущности или объекты-значения.

Таким образом, агрегаты могут сохраняться также с использованием JPA. Все операции сохранения выполняются для всего агрегата, распространяясь от корневого объекта на остальные объекты, принадлежащие агрегату. Как мы увидим в последующих подразделах, JPA поддерживает сохранение сложных иерархий объектов.

Репозитории

Для доступа к базе данных в концепции проблемно-ориентированного проектирования определены репозитории, управляющие сохранением и согласованностью сущностей. Причиной появления репозиториев стало желание иметь единую точку ответственности, которая позволяет модели предметной области сохранять устойчивость и согласованность. Описание этих функциональных возможностей не должно загромождать код модели предметной области деталями реализации. Поэтому в проблемно-ориентированном проектировании определена концепция репозиториев, которые самостоятельно и согласованно инкапсулируют эти операции.

Репозитории являются точками входа для операций сохранения сущностей определенного типа. Поскольку необходимо идентифицировать только экземпляры агрегатов и сущностей, лишь эти типы нуждаются в репозиториях.

В Java EE и JPA уже реализована функциональность, которая хорошо соответствует идее репозиториев, — это диспетчер сущностей `EntityManager` из JPA. Он используется для сохранения и извлечения объектов, которые определены как сущности, а также управления такими объектами или их потенциальными иерархиями. То, что объекты, управляемые JPA, должны быть идентифицируемыми сущностями, идеально соответствует ограничениям, установленным идеей сущностей в концепции проблемно-ориентированного проектирования.

Диспетчер сущностей внедряется в управляемые объекты и действует в них. Это соответствует представлению о том, что сервисы, будь то контуры или элементы управления, предназначены для построения бизнес-сценариев, в данном случае путем вызова диспетчера для сохранения сущностей.

Фабрики

Фабрики в проблемно-ориентированном проектировании используются потому, что при создании объектов предметной области часто применяются более сложные логика и ограничения, чем простой вызов конструктора. Для создания согласованных объектов предметной области могут потребоваться валидация и другие сложные процедуры. Поэтому имеет смысл выделить логику создания в особые методы или классы, которые инкапсулируют ее от остального приложения.

Фабрики создаются по тем же причинам, что и рассмотренные ранее шаблоны проектирования «Абстрактная фабрика» и «Метод фабрики». Поэтому здесь используется такая же реализация с применением функций CDI. Спецификация CDI — это, в сущности, и есть функциональность фабрики.

Объекты-фабрики из предметной области также могут быть реализованы как методы в составе другого класса модели предметной области, такого как сущность. Эти решения реализуются на чистом Java, без каких-либо фреймворков или аннотаций. Функциональность водительского журнала, рассмотренная в разделе, посвященном шаблону проектирования «Метод фабрики», — хороший пример метода фабрики, включенного в объект предметной области. Если класс из предметной области самостоятельно выполняет всю логику, вполне разумно включить туда и логику фабрики.

События предметной области

События предметной области представляют собой события, относящиеся к бизнес-сценариям. Обычно они генерируются бизнес-сценариями и имеют семантику, соответствующую предметной области. Примерами событий предметной области являются `UserLoggedIn`, `ACticlePurchased` и `CoffeeBrewFinished`.

События предметной области обычно представляют в виде объектов-значений, содержащих требуемую информацию. В Java такие события реализованы в виде неизменяемых POJO-объектов. События, произошедшие в прошлом, не могут быть впоследствии изменены, поэтому настоятельно рекомендуется делать эти

объекты неизменяемыми. Как мы уже видели, для публикации событий со слабым связыванием и наблюдения за ними можно задействовать функциональность событий CDI. В CDI любые типы Java могут быть использованы для публикации в виде событий. Таким образом, концепция событий предметной области является скорее бизнес-определением, чем техническим термином.

События предметной области особенно важны для порождения событий и событийно-ориентированных архитектур, которые будут подробно рассмотрены в главе 8.

Внешняя и сквозная функциональность в корпоративных приложениях

Итак, мы рассмотрели концепции и реализации, необходимые для построения логики предметной области приложения. Теоретически этого достаточно для разработки бизнес-логики приложения, однако от бизнес-сценариев мало пользы, если они недоступны извне системы. Поэтому рассмотрим технически обоснованные внешние и сквозные функции приложений. Несмотря на то что эти функциональные возможности не входят в ядро предметной области, их все же необходимо реализовать. Примерами технически обоснованных задач приложения являются доступ к внешним системам и базам данных, настройка приложения и кэширование.

Обмен данными с внешними системами

Обмен данными с внешним миром является одной из важнейших технических задач корпоративного приложения. Без такой коммуникации приложение вряд ли принесет какую-либо пользу клиенту.

Выбор технологии связи

Когда корпоративным системам требуется связь, возникает вопрос: какие коммуникационные протоколы и технологии для этого применить? Существует множество форм синхронного и асинхронного обмена данными. Прежде чем углубиться в эту тему, необходимо ответить на ряд вопросов.

Какая коммуникационная технология поддерживается выбранными языками и фреймворками? Используются ли какие-то готовые системы, требующие определенной формы обмена данными? Как системы передают информацию — синхронно или асинхронно? С какими решениями знакома команда инженеров, работающих над проектом? Должна ли система действовать в среде, где важна высокая производительность?

Вновь взглянув на продукт с точки зрения бизнеса, мы поймем, что связь между системами необходима, но она *не должна мешать* реализации бизнес-сценариев.

Другими словами, обмен информацией должен быть реализован максимально простым способом, соответствующим данной предметной области, независимо от того, является коммуникация синхронной или асинхронной. Эти соображения значительно влияют не только на фактическую реализацию, но и на то, соответствует ли весь сценарий использования выбранному решению. Таким образом, это один из первых вопросов, которые необходимо задать независимо от того, как организована связь. Синхронная коммуникация обеспечивает согласованность и последовательность обмена информацией. Однако ее скорость меньше, чем у асинхронной коммуникации, и ее нельзя масштабировать бесконечно. Асинхронный обмен данными ослабляет связь между задействованными системами и повышает общую производительность, но одновременно увеличивает накладные расходы. Он также позволяет реализовать сценарии, в которых нет постоянного надежного доступа ко всем системам. Из соображений простоты, а также для обеспечения согласованности корпоративные приложения обычно применяют синхронную связь.

Выбранный способ обмена данными должен поддерживаться не только языком и фреймворками, но и используемыми средами и инструментами. Необходимо выяснить, устанавливают ли среда и свойства сети какие-либо ограничения на связь? В сущности, именно это стало одной из причин того, что протокол SOAP был широко распространен в прошлом: он действовал через сетевой порт 80, что допускалось в большинстве сетевых конфигураций. Еще одним важным фактором является поддержка инструментария разработки и отладки. Именно это стало причиной популярности HTTP.

В системах Java поддерживается большинство коммуникационных технологий — либо изначально, например HTTP, либо сторонними библиотеками. Что, конечно, не относится к другим технологиям. Именно это является одной из проблем, связанных с протоколом SOAP. Поддержка протокола SOAP была реализована только в приложениях Java и .NET. В других технологиях предпочтение отдавалось иным способам обмена данными.

На производительность коммуникационных технологий важно обращать внимание не только в высокопроизводительных средах. Обмен информацией по сети в ходе коммуникаций между процессами или внутри процессов всегда приводит к огромным накладным расходам. Вопрос только в том, насколько они велики. В основном это касается плотности информации, скорости обработки сообщений и другой полезной нагрузки. В каком формате передается информация — двоичном или обычном текстовом? В каком формате отображается содержимое? Вообще говоря, двоичные форматы с высокой плотностью и низким дублированием информации более эффективны и передают данные меньшего размера, но их сложнее отлаживать.

Еще одним важным свойством является гибкость коммуникационных решений. Выбранная технология не должна чересчур ограничивать обмен информацией. В идеале протокол должен поддерживать разные способы коммуникации, например синхронную и асинхронную связь, двоичные форматы и Hypermedia.

Поскольку основной задачей приложения является реализация бизнес-логики, выбранная технология должна идеально соответствовать общим требованиям, предъявляемым к программному продукту.

В современных системах чаще всего используется протокол связи HTTP. На то есть несколько причин. HTTP хорошо поддерживается большинством языковых платформ, фреймворков и библиотек. Для этого протокола создано множество разнообразных инструментов, и он хорошо известен большинству инженеров-программистов. HTTP не накладывает особых ограничений на способ использования и поэтому может применяться для всех видов обмена информацией — синхронной и асинхронной связи, Hypermedia и прямых вызовов удаленных функций. Однако HTTP поощряет определенный способ использования. В следующем разделе мы обсудим семантику HTTP, вызовы удаленных процедур и REST.

Существуют протоколы связи, которые не всегда, но в большинстве случаев применяются поверх HTTP. В прошлом самым ярким примером такого протокола был SOAP, более свежий пример — gRPC. Оба они реализуют вызов удаленных процедур. Вызовы удаленных процедур представляют собой вариант непосредственного вызова функции из другой системы по сети. В описании функции указываются ее входные и выходные значения. В SOAP такие вызовы удаленных процедур реализовались в формате XML, в gRPC используются двоичные буферы протокола для сериализации структур данных.

В отношении синхронного или асинхронного обмена данными настоятельно рекомендуется последовательно придерживаться выбранного поведения, зависящего от требований бизнеса. В общем случае не следует смешивать синхронное и асинхронное поведение. Синхронная передача данных между сервисами, которые содержат асинхронную логику, не имеет смысла. Вызывающий абонент будет заблокирован до завершения асинхронного процесса, и вся функциональность не станет масштабироваться. Но иногда имеет смысл задействовать асинхронную связь для инкапсуляции длительных синхронных процессов. Например, это могут быть внешние системы, которые нельзя изменить, или устаревшие приложения. Клиентский компонент может подключаться к такой системе в отдельном потоке, позволяя вызывающему потоку продолжить работу сразу после вызова. Клиентский поток либо блокируется до завершения синхронного процесса, либо использует опрос. Однако предпочтительнее моделировать системы и способ связи после того, как будут удовлетворены все бизнес-требования.

Существует немало протоколов и коммуникационных форматов, многие из которых являются коммерческими продуктами. Инженерам рекомендуется разбираться в различных концепциях и способах связи в целом. Коммуникационные технологии могут изменяться, но принципы обмена данными остаются прежними. На момент написания этой книги наиболее популярным протоколом связи был HTTP. Это, возможно, одна из самых важных технологий — она хорошо проработана и имеет отличную инструментальную поддержку.

Синхронный обмен данными по HTTP

Сегодня синхронная связь в корпоративных системах реализуется в основном через протокол HTTP. Корпоративные приложения предоставляют конечные HTTP-точки, к которым обращаются клиенты. Эти точки обычно имеют вид веб-сервисов или клиентских веб-интерфейсов в формате HTML с передачей данных через HTTP.

Существуют различные способы разработки и описания веб-сервисов. В простейшем случае требуется всего лишь вызвать функцию из другой системы. Эта функция должна быть описана с указанием ее входных и выходных значений. Такие функции, или *вызовы удаленных процедур (remote procedure calls, RPC)*, в данном случае реализуются через HTTP, обычно с использованием формата XML для описания аргументов. Во времена J2EE эти типы веб-сервисов были довольно широко распространены, наиболее ярким примером был протокол SOAP, реализованный на базе стандарта JAX-WS. Однако SOAP и его формат XML были очень громоздкими и не поддерживались другими языками, кроме Java и .NET.

В современных системах гораздо чаще применяется архитектурный стиль REST с его концепцией и ограничениями.

Передача состояния представления

Идеи и ограничения *передачи состояния представления (Representational State Transfer, REST)*, показанной Роем Т. Филдингом (Roy T. Fielding), формируют архитектурный стиль веб-сервисов, который удовлетворяет потребности большинства корпоративных приложений. Его идеи приводят к появлению систем, более тесно связанных с интерфейсами, предоставляющими единообразный и простой доступ для разных клиентов.

Ограничение *унифицированного интерфейса* в REST требует, чтобы ресурсы идентифицировались в запросах по их URI в веб-системах. Ресурсы представляют собой объекты предметной области, например пользователей или статьи, причем у каждого из этих объектов в корпоративном приложении есть свой URL. Другими словами, URL-адреса больше не являются методами RPC, а становятся действительными сущностями предметной области. Эти представления модифицируются единообразным способом в HTTP с использованием таких HTTP-методов, как `GET`, `POST`, `DELETE`, `PATCH` и `PUT`. Сущности могут быть представлены в разных форматах, запрашиваемых клиентом, таких как XML или JSON. Если эти форматы поддерживаются сервером, то клиенты могут выбирать, в каком формате обращаться к конкретному пользователю — XML или JSON.

Еще одной особенностью унифицированного ограничения интерфейса является применение Hypermedia в качестве механизма состояния приложения. Hypermedia означает соединение взаимосвязанных ресурсов с помощью гиперссылок. Передаваемые клиенту ресурсы REST могут содержать ссылки на другие, семантически связанные с ними ресурсы. Если пользователь включает информацию

об их диспетчере, то эту информацию можно сериализовать, используя ссылку на ресурс второго пользователя (диспетчера).

Далее приведен пример представления книги со ссылками Hypermedia, включенными в ответ JSON:

```
{
    "name": "Java",
    "author": "Duke",
    "isbn": "123-2-34-456789-0",
    "_links": {
        "self": "https://api.example.com/books/12345",
        "author": "https://api.example.com/authors/2345",
        "related-books": "https://api.example.com/books/12345/related"
    }
}
```

На сайтах с различной информацией для пользователей эти ссылки особенно важны. В API Hypermedia они применяются REST-клиентами для навигации по API. Концепция открытости ослабляет зависимость и увеличивает способность к эволюции задействованных систем. Если эта концепция реализована полностью, то клиенты должны знать только точку входа API, а доступ ко всем ресурсам осуществляется по семантическим ссылкам, таким как `related-books`. Они отражают известные взаимосвязи, используя предоставленные URL.

В большинстве API REST клиентам недостаточно только отслеживать ссылки, чтобы получать доступ к ресурсам методом HTTP `GET`. Обмен информацией осуществляется с использованием методов HTTP, таких как `POST` и `PUT`, изменяющих состояние и тело запроса. Hypermedia также поддерживает эти так называемые действия (actions), задействуя свои элементы управления. Действия описывают не только целевой URL, но и метод HTTP, и информацию, которую необходимо передать.

Далее приведен сложный пример использования концепции действий в Hypermedia. Здесь продемонстрирован тип содержимого Siren. Цель этого примера — дать вам представление о возможном содержимом ответов Hypermedia:

```
{
    "class": [ "book" ],
    "properties": {
        "isbn": "123-2-34-456789-0",
        "name": "Java",
        "author": "Duke",
        "availability": "IN_STOCK",
        "price": 29.99
    }
    "actions": [
        {
            "name": "add-to-cart",
            "title": "Поместить книгу в корзину",
            "method": "POST",
            "href": "http://api.example.com/shopping-cart",
```

```
            "type": "application/json",
            "fields": [
                { "name": "isbn", "type": "text" },
                { "name": "quantity", "type": "number" }
            ]
        }
    ],
    "links": [
        { "rel": [ "self" ], "href": "http://api.example.com/books/1234" }
    ]
}
```

Это пример типа содержимого, поддерживающего элементы управления Hypermedia. На момент написания книги ни один из типов содержимого с поддержкой Hypermedia, таких как Siren, HAL или JSON-LD, не стал фактическим стандартом. Однако тип содержимого Siren в достаточной степени поддерживает концепции обмена данными с помощью ссылок и действий.

Использование Hypermedia отделяет клиента от сервера. Прежде всего, только сервер отвечает за URL-адреса. Клиенты не имеют представления о том, как создаются URL, например, что книга находится по адресу `/books/1234`, который состоит из пути `/books/` и идентификатора книги. В реальных проектах нам часто встречались случаи дублирования логики URL-адресов на стороне клиента.

Кроме того, отделена возможность изменения состояния на сервере. Например, инструкция, согласно которой клиенты должны получать содержимое `/shopping-cart` методом `POST` в формате JSON определенной структуры, больше не записана жестко на стороне клиента, а формируется динамически. Клиент будет только ссылаться на действие Hypermedia, используя отношение или имя (в данном случае `add-to-cart`) и информацию, предоставленную в действии. При этом подходе он должен знать только смысл действия «добавить в корзину» (add-to-cart) и источники данных — поля ISBN и количества единиц товара, что относится к логике клиента. Значения полей можно получить непосредственно из представления ресурса или клиентского процесса. Например, количество книг может иметь в пользовательском интерфейсе вид раскрывающегося списка.

Еще одна потенциальная возможность использования Hypermedia — отделить бизнес-логику от клиента, задействуя ссылки и действия, и направить клиента на доступные ресурсы. Информация, содержащаяся в доступных ссылках и действиях, применяется для неявного указания клиентам, какие бизнес-сценарии возможны в текущем состоянии системы. Предположим, что в корзину можно помещать только книги, имеющиеся на складе. Тогда клиенты, которые реализуют это поведение, то есть показывают кнопку Добавить в корзину, должны знать об этой логике. В этом случае клиентская функция будет проверять, соответствует ли доступность книги заданным критериям и т. д. Технически эта бизнес-логика должна находиться только на стороне сервера. Предоставляя динамически ссылки и действия доступным ресурсам, сервер определяет, какие функции возможны

в текущем состоянии. Добавление в корзину будет доступным только в том случае, если книга действительно может быть туда помещена. Таким образом, логика клиента упрощается и сводится к проверке того, доступны ли ссылки и действия с известными отношениями или именами. В итоге клиент показывает активную кнопку Добавить в корзину, только если в результате ее нажатия может быть выполнено соответствующее действие.

После появления Java EE архитектурный стиль REST привлекает все больше внимания. Хотя большинство веб-сервисов Java EE не реализуют все ограничения, определяемые архитектурным стилем REST, особенно Hypermedia, в целом их можно считать REST-сервисами.

Для получения дополнительной информации об ограничениях REST можно обратиться к диссертации Роя Т. Филдинга (Roy T. Fielding) «Архитектурные стили и разработка сетевых приложений» (Architectural Styles and the Design of Network-based Software Architectures).

Java API для веб-сервисов RESTful

В Java EE *API для веб-сервисов RESTful (JAX-RS)* используются для определения REST-сервисов и доступа к ним. JAX-RS широко применяются в экосистеме Java, в том числе и в других корпоративных технологиях. Программисты отдают предпочтение декларативной модели разработки, которая значительно упрощает создание REST-сервисов.

Так называемые ресурсы JAX-RS — это REST-ресурсы, доступные по определенному URL. Ресурсы JAX-RS — это методы в классе ресурсов, реализующие бизнес-логику после обращения по URL-адресу с помощью какого-либо метода HTTP. Рассмотрим пример класса ресурса JAX-RS для пользователей:

```
import javax.ws.rs.Path;
import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("users")
@Produces(MediaType.APPLICATION_JSON)
public class UsersResource {

    @Inject
    UserStore userStore;

    @GET
    public List<User> getUsers() {
        return userStore.getUsers();
    }
}
```

Метод `getUsers()` — это метод ресурсов JAX-RS, который будет вызываться контейнером после того, как клиент выполнит HTTP-вызов `GET .../users`. Затем список пользователей возвращается клиенту в формате JSON — в виде

JSON-массива, содержащего JSON-объекты для каждого из пользователей. Это определено в аннотации `@Produces`, которая тут неявно использует *Java API for JSON Binding (JSON-B)* для преобразования типов Java в соответствующие представления JSON.

Здесь мы видим, как работает принцип инверсии управления. Нам не приходится самостоятельно подключать или регистрировать URL — достаточно объявления с применением аннотации `@Path`. То же самое верно для преобразования типов Java в такие представления, как JSON. Достаточно указать в декларативной форме, какие форматы представления мы хотим предоставить. Остальное обрабатывается контейнером. Реализация JAX-RS также обеспечивает необходимую HTTP-коммуникацию. Возвращая объект, в данном случае список пользователей, JAX-RS неявно предполагает, что код состояния HTTP — `200 OK`, который возвращается клиенту вместе с представлением JSON.

Для того чтобы зарегистрировать ресурсы JAX-RS в контейнере, приложение может отправить подкласс класса `Application`, который запускает среду выполнения JAX-RS. После аннотирования этого класса с помощью `@ApplicationPath` предоставляемый путь автоматически регистрируется как Servlet. Далее показан класс конфигурации JAX-RS, которого достаточно для подавляющего большинства сценариев использования:

```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("resources")
public class JAXRSConfiguration extends Application {
    // дополнительная настройка не требуется
}
```

JAX-RS и другие стандарты, применяемые в Java EE, используют принцип программирования по соглашениям. Стандартное поведение этого REST-ресурса приемлемо для большинства сценариев. Если же это не так, то стандартное поведение всегда можно переопределить с помощью настраиваемой логики. Именно по этой причине JAX-RS, кроме прочего, поддерживает продуктивную модель программирования. Стандартные сценарии можно реализовать очень быстро, причем с возможностью дальнейшего улучшения.

Рассмотрим более полный пример. Предположим, что мы хотим создать нового пользователя в системе, в которой клиент использует наш REST-сервис. В соответствии с HTTP-семантикой это действие будет POST-запросом к ресурсу пользователя, поскольку создается новый ресурс, который еще не может быть идентифицирован. Разница между методами POST и PUT заключается в том, что последний может все, только изменяя доступный ресурс с предложенным представлением, тогда как POST может создавать новые ресурсы в виде новых URL, что и требуется в данном случае. Мы создаем нового пользователя, который будет идентифицироваться по новому, сгенерированному URL. Если ресурс для нового пользователя уже создан, клиент должен быть направлен на этот URL.

Создание ресурсов обычно реализуется с кодом состояния `201 Created`, указывающим, что новый ресурс создан успешно, и заголовком `Location`, содержащим URL данного ресурса.

Чтобы выполнить это требование, нужно предоставить в ресурсе JAX-RS дополнительную информацию. Далее показано, как это делается в методе `createUser()`:

```
import javax.ws.rs.Consumes;
import javax.ws.rs.PathParam;
import javax.ws.rs.POST;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;

@Path("users")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class UsersResource {

    @Inject
    UserStore userStore;

    @Context
    UriInfo uriInfo;

    @GET
    public List<User> getUsers() {
        return userStore.getUsers();
    }

    @GET
    @Path("{id}")
    public User getUser(@PathParam("id") long id) {
        return userStore.getUser(id);
    }

    @POST
    public Response createUser(User user) {
        long id = userStore.create(user);

        URI userUri = uriInfo.getBaseUriBuilder()
            .path(UsersResource.class)
            .path(UsersResource.class, "getUser")
            .build(id);

        return Response.created(userUri).build();
    }
}
```

Мы воспользовались входящей в состав JAX-RS функцией `UriInfo`, поэтому нам не нужно повторяться при построении нового URL. Эта функция берет информацию о пути, которая уже есть в аннотации класса ресурсов. Метод `Response` применяется для того, чтобы указать фактический ответ HTTP с использованием

шаблона «Строитель». JAX-RS отмечает, что возвращаемый тип нашего метода теперь является спецификацией ответа, и будет отвечать клиенту соответствующим образом. Благодаря такому подходу обеспечиваются полный контроль и гибкость представления ответа для клиента.

Как видите, эти методы являются точкой входа для бизнес-сценариев. Мы внедряем контур `UserStore`, который в данном случае реализуется как EJB-объект, обеспечивающий логику для возврата списка пользователей и создания новых пользователей.

JAX-RS предоставляет эффективный и простой способ реализации бизнес-функций с помощью веб-сервисов RESTful. Программистам не нужно писать низкоуровневый связующий код для HTTP, если достаточно стандартного поведения.

Преобразование типов содержимого HTTP

Все с той же целью — обеспечить максимальную производительность работы программистов — в Java EE включены стандарты прозрачного преобразования POJO-объектов в JSON или XML. В рассмотренном примере с JAX-RS был неявно использован JSON-B для преобразования типов `User` в объекты и массивы JSON.

Здесь снова задействуется принцип программирования по соглашениям. Если ничего специально не указано, то JSON-B предполагает преобразование свойств POJO непосредственно в пары «ключ — значение» JSON-объекта. Идентификатор пользователя также присутствует в выходном массиве JSON.

Это справедливо и для *Java Architecture for XML Binding (JAXB)* и соответствующей привязки XML, которая была включена в Java EE намного раньше, чем JSON-B. Оба стандарта поддерживают принцип декларативной конфигурации с использованием аннотаций, которые размещаются в преобразуемых типах Java. Для того чтобы изменить представление типа в JSON, нужно добавить аннотации в соответствующие поля:

```
import javax.json.bind.annotation.JsonbProperty;
import javax.json.bind.annotation.JsonbTransient;

public class User {

    @JsonbTransient
    private long id;

    @JsonbProperty("username")
    private String name;

    …
}
```

Реализовать более сложное преобразование ресурсов, как в приведенных ранее примерах с книгами и Hypermedia, можно, используя декларативное преобра-

зование. Например, преобразовать ссылки в ресурс книг можно с применением карты преобразования, содержащей ссылки и их соответствия:

```
public class Book {

    @JsonbTransient
    private long id;

    private String name;
    private String author;
    private String isbn;

    @JsonbProperty("_links")
    private Map<String, URI> links;

    …
}
```

Эти ссылки заданы в ресурсе JAX-RS таким образом:

```
@Path("books")
@Produces(MediaType.APPLICATION_JSON)
public class BooksResource {

    @Inject
    BookStore bookStore;

    @Context
    UriInfo uriInfo;

    @GET
    public List<Book> getBooks() {
        List<Book> books = bookStore.getBooks();
        books.forEach(this::addLinks);
        return books;
    }

    @GET
    @Path("{id}")
    public Book getBook(@PathParam("id") long id) {
        Book book = bookStore.getBook(id);
        addLinks(book);
        return book;
    }

    private void addLinks(Book book) {
        URI selfUri = uriInfo.getBaseUriBuilder()
            .path(BooksResource.class)
            .path(BooksResource.class, "getBook")
            .build(book.getId());

        book.getLinks().put("self", selfUri);
        // другие ссылки
    }
}
```

Вывод списка книг будет выглядеть следующим образом:

```
[
    {
        "name": "Java",
        "author": "Duke",
        "isbn": "123-2-34-456789-0",
        "_links": {
            "self": "https://api.example.com/books/12345",
            "author": "https://api.example.com/authors/2345",
            "related-books": "https://api.example.com/books/12345/related"
        }
    },
    …
]
```

Придерживаясь этого принципа, можно программно создавать ссылки с отношениями, по которым переходит клиент. Однако, используя подход Hypermedia, мы довольно быстро достигнем того, что декларативное преобразование станет требовать слишком больших накладных расходов на модель. Карта ссылок и отношений не является частью предметной области — это техническая необходимость, которая должна подвергаться сомнению. Мы могли бы ввести типы транспортных объектов и таким образом отделить техническое преобразование от модели предметной области. Но это неизбежно привело бы к многократному дублированию и засорению проекта большим количеством классов, не приносящих пользы бизнесу.

Еще одна проблема, с которой мы сталкиваемся, — то, что для Hypermedia необходима гибкость. Даже в более простых случаях использования элементов управления Hypermedia приходится создавать ссылки и действия в зависимости от текущего состояния системы. Hypermedia свойственно контролировать поток клиентов и направлять его на определенные ресурсы. Например, ответ клиента должен содержать действие для размещения заказа только в том случае, если книга есть на складе, а у пользователя на счету достаточно средств. Для этого требуется, чтобы ответ преобразовывался в зависимости от ситуации. Поскольку во время выполнения приложения нелегко достичь декларативного преобразования, необходим более гибкий подход.

Еще в Java EE 7 появился стандарт *Java API for JSON Processing (JSON-P)*, описывающий программное преобразование структур JSON в стиле шаблона «Строитель». Можно просто вызывать конструкторы `JsonObjectBuilder` или `JsonArrayBuilder` и создавать структуры любой сложности:

```
import javax.json.Json;
import javax.json.JsonObject;
…

JsonObject object = Json.createObjectBuilder()
    .add("hello", Json.createArrayBuilder()
        .add("hello")
        .build())
    .add("key", "value")
    .build();
```

В результате получится такой JSON-объект:

```
{
    "hello": [
        "hello"
    ],
    "key": "value"
}
```

Этот подход особенно полезен в ситуациях, когда нужна большая гибкость, например в Hypermedia. Стандарт JSON-P, подобно JSON-B и JAXB, легко интегрируется с JAX-RS. Методы ресурсов JAX-RS, возвращающие типы JSON-P, такие как `JsonObject`, автоматически возвращают содержимое в формате JSON вместе с соответствующим ответом; никакой дополнительной настройки не требуется. Рассмотрим пример, содержащий ссылки на ресурсы и реализованный с помощью JSON-P:

```
import javax.json.JsonArray;
import javax.json.stream.JsonCollectors;

@Path("books")
public class BooksResource {

    @Inject
    BookStore bookStore;

    @Context
    UriInfo uriInfo;

    @GET
    public JsonArray getBooks() {
        return bookStore.getBooks().stream()
            .map(this::buildBookJson)
            .collect(JsonCollectors.toJsonArray());
    }

    @GET
    @Path("{id}")
    public JsonObject getBook(@PathParam("id") long id) {
        Book book = bookStore.getBook(id);
        return buildBookJson(book);
    }

    private JsonObject buildBookJson(Book book) {
        URI selfUri = uriInfo.getBaseUriBuilder()
                .path(BooksResource.class)
                .path(BooksResource.class, "getBook")
                .build(book.getId());

        URI authorUri = …

        return Json.createObjectBuilder()
                .add("name", book.getName())
                .add("author", book.getName())
                .add("isbn", book.getName())
```

```
                .add("_links", Json.createObjectBuilder()
                .add("self", selfUri.toString())
                    .add("author", authorUri.toString()))
                .build();
    }
}
```

Объекты JSON-P создаются динамически с использованием шаблона «Строитель». Он позволяет создавать на выходе любые структуры. Применять JSON-P также рекомендуется в тех случаях, когда для связи требуется одно представление сущности, а для текущей модели — другое. В прошлом для этой цели в состав проектов всегда входили объекты переноса данных, или DTO. Теперь объекты JSON-P фактически заменили собой объекты переноса данных. Благодаря этому подходу нам уже не нужен еще один класс, дублирующий большинство структур модельной сущности.

Однако в этом примере тоже есть некоторое дублирование. Имена свойств получившихся JSON-объектов теперь представлены в виде строк. Для того чтобы немного оптимизировать код примера, создадим единую точку ответственности — управляемый объект, который будет создавать все объекты JSON-P из модельных сущностей.

Этот объект — назовем его `EntityBuilder` — будет внедрен в данный и другие классы ресурсов JAX-RS. Это не избавит нас от дублирования, но оно будет инкапсулировано в единую точку ответственности и станет многократно использоваться разными классами ресурсов. В следующем коде показан пример объекта `EntityBuilder` для книг и, возможно, других объектов для преобразования в формат JSON:

```
public class EntityBuilder {

    public JsonObject buildForBook(Book book, URI selfUri) {
        return Json.createObjectBuilder()
                …
    }
}
```

Если представление, необходимое для какой-либо конечной точки или внешней системы, отличается от нашей модели, то мы не сможем полностью избежать дублирования, не получив взамен других недостатков. Используя этот подход, мы отделяем логику преобразования от модели и получаем полную гибкость. Свойства POJO-объектов преобразуются путем вызова шаблона «Строитель». По сравнению с внедрением отдельных классов объектов переноса и их преобразованием в другой функционал это позволяет получить более простой и, в конечном счете более компактный код.

Но вернемся к примеру с Hypermedia, снова используя действия Siren по добавлению товара в корзину. Этот пример дает представление о возможном содержимом ответов Hypermedia. Для таких ответов результат должен быть дина-

мичным и гибким и зависеть от состояния приложения. Теперь мы представляем себе гибкость и возможности программного преобразования, такого как JSON-P. Этот выходной объект невозможно получить, если задействовать декларативное преобразование POJO, — попытка привела бы к созданию довольно сложного графа объектов. В Java EE для такого содержимого рекомендуется применить JSON-P как единую точку ответственности или стороннюю зависимость.

Для преобразования объектов Java в структуры данных JSON или XML стандарты JAXB, JSON-B и JSON-P предлагают полную интеграцию с другими стандартами Java EE, такими как JAX-RS. Наряду с продемонстрированной ранее интеграцией в JAX-RS можно также интегрировать CDI-внедрение. Эта совместимость сохраняется для всех современных стандартов Java EE.

Адаптеры типа JSON-B позволяют также преобразовывать нестандартные типы Java, неизвестные в JSON-B. Такие типы Java конвертируются в известные и преобразуемые типы. Типичный пример — сериализация ссылок на объекты как идентификаторы:

```
import javax.json.bind.annotation.JsonbTypeAdapter;

public class Employee {

    @JsonbTransient
    private long id;
    private String name;
    private String email;

    @JsonbTypeAdapter(value = OrganizationTypeAdapter.class)
    private Organization organization;

    …
}
```

Адаптер типа, указанный в поле `organization`, используется для представления ссылки как идентификатора организации. Для разрешения этой ссылки нужно найти приемлемые организации. Такую функциональность можно просто внедрить в адаптер типа JSON-B:

```
import javax.json.bind.adapter.JsonbAdapter;

public class OrganizationTypeAdapter implements JsonbAdapter<Organization,String> {

    @Inject
    OrganizationStore organizationStore;

    @Override
    public String adaptToJson(Organization organization) {
        return String.valueOf(organization.getId());
    }

    @Override
```

```
    public Organization adaptFromJson(String string) {
        long id = Long.parseLong(string);
        Organization organization = organizationStore.getOrganization(id);

        if (organization == null)
            throw new IllegalArgumentException("Организации с таким ID
                      не существует" + string);

        return organization;
    }
}
```

На этом примере показано преимущество применения нескольких стандартов, хорошо совместимых друг с другом. Программисты могут просто использовать и интегрировать их функциональные возможности, не тратя времени на настройку и разработку *связующего* программного обеспечения.

Валидация запросов

JAX-RS позволяет интегрировать в систему конечные точки HTTP. Это подразумевает, в частности, преобразование запросов и ответов в типы Java, применяемые в приложении. Однако клиентские запросы необходимо валидировать, чтобы предотвратить неправильное использование системы.

Стандарт *Bean Validation* обеспечивает всестороннюю валидацию. Смысл ее состоит в том, чтобы объявить условия валидации, например «ненулевое поле», «неотрицательное целое число» или «повышение зарплаты должно соответствовать политике компании», и связать их с типами и свойствами Java. Самые распространенные технически обоснованные условия уже интегрированы в стандарт. К ним можно добавить специальные условия, особенно те, что вызваны бизнес-функциональностью и бизнес-валидацией. Это интересно не только с технической точки зрения, но и с точки зрения предметной области. Логика валидации, обусловленной предметной областью, также может быть реализована с помощью этого стандарта.

Валидация активизируется аннотированием параметров метода, возвращаемых типов и свойств с помощью `@Valid`. Валидация может применяться в различных местах приложения, но она особенно важна в конечных точках. Если для параметра ресурсного метода JAX-RS указать аннотацию `@Valid`, то будет выполняться валидация тела или параметра запроса. Если запрос не пройдет валидацию, то JAX-RS автоматически ответит на HTTP-запрос кодом состояния, указывающим на ошибку клиента.

В следующем примере показана интеграция валидации пользователя:

```
import javax.validation.Valid;
import javax.validation.constraints.NotNull;

@Path("users")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
```

```
public class UsersResource {

    …
    @POST

    public Response createUser(@Valid @NotNull User user) {
        …
    }
}
```

Тип пользователя снабжен аннотацией с условиями валидации:

```
import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;

public class User {

    @JsonbTransient
    private long id;

    @NotBlank
    private String name;

    @Email
    private String email;

    …
}
```

Аннотации, размещенные в методе JAX-RS, вызывают валидацию тела запроса, как только он поступит от клиента. Тело запроса должно быть доступным, не равняться `null` и соответствовать конфигурации типа пользователя. Свойство имени пользователя не должно быть пустым — другими словами, оно не должно быть `null` или содержать только пробелы. Свойство `email` должно соответствовать формату адреса электронной почты. Эти условия учитываются при проверке объекта пользователя.

Валидацию объектов проводит метод `Validator`, входящий в состав Bean Validation. Если проверка не пройдена, валидатор вызывает исключение `ConstraintViolationException`. Функция валидатора также может быть доступна, если внедрить зависимость и вызвать ее программно. JAX-RS вызывает валидатор автоматически и, если проверка не пройдена, отправляет клиенту соответствующий ответ.

В рассмотренном примере не удастся выполнить неразрешенные вызовы HTTP POST для ресурса `/users/`, такие как создание пользовательских представлений без имени. Это приведет к выдаче статуса запроса `400 Bad Request`, как принято в JAX-RS по умолчанию, если валидация клиента завершилась неудачей.

Если клиенту требуется дополнительная информация о том, почему запрос был отклонен, поведение по умолчанию может быть расширено. Исключения, генерируемые валидатором при ошибках валидации, могут преобразовываться

в ответы HTTP с помощью функции преобразования JAX-RS. Преобразователи исключений обрабатывают исключения, сгенерированные методами ресурсов JAX-RS, в соответствующие ответы клиентам. Далее показан такой преобразователь `ExceptionMapper` для исключения `ConstraintViolationExceptions`:

```
import javax.validation.ConstraintViolationException;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;

@Provider
public class ValidationExceptionMapper implements
ExceptionMapper<ConstraintViolationException> {

    @Override
    public Response toResponse(ConstraintViolationException exception) {
        Response.ResponseBuilder builder =
            Response.status(Response.Status.BAD_REQUEST);

        exception.getConstraintViolations()
                .forEach(v -> {
                    builder.header("Error-Description", ...);
                });
        return builder.build();
    }
}
```

Преобразователи исключений являются поставщиками для JAX-RS в процессе выполнения приложения. Поставщики настраиваются либо программно в базовом классе приложения JAX-RS, либо, как показано здесь, декларативным способом с использованием аннотации `@Provider`. В ходе этого JAX-RS проверяет классы на наличие поставщиков и автоматически их применяет.

Преобразователь исключений регистрируется для каждого типа исключений и его подтипов. Все исключения нарушения условий валидации, генерируемые ресурсным методом JAX-RS, преобразуются в ответы клиенту, которые включают в себя базовое описание того, какие поля вызвали ошибку при валидации. Сообщения о нарушениях предоставляются функциями Bean Validation в виде, понятном человеку.

Если встроенных условий недостаточно для валидации, можно ввести дополнительные. Это особенно полезно для правил валидации, специфических для данной предметной области. Например, имена пользователей могут потребовать более сложной валидации, основанной на текущем состоянии системы. В следующем примере имена пользователей не должны приниматься при создании новых пользователей. Разумеется, вместе с ними на формат объекта и допустимые символы могут накладываться и другие ограничения:

```
public class User {

    @JsonbTransient
    private long id;

    @NotBlank
```

```
    @UserNameNotTaken
    private String name;

    @Email
    private String email;
    …
}
```

Аннотация `@UserNameNotTaken` — это специальное условие валидации, определенное приложением. Условия валидации делегируют управление валидатору — классу, который и выполняет настоящую проверку. Валидаторы имеют доступ к аннотированному объекту, такому как класс или в данном случае поле. Специальная функция проверяет, удовлетворяет ли данный объект всем условиям валидации. Для контроля нестандартных нарушений вывода сообщений и дополнительной информации в методе валидации может использоваться `ConstraintValidatorContext`.

В следующем примере показано определение нестандартного условия валидации:

```
import javax.validation.Constraint;
import javax.validation.Payload;

@Constraint(validatedBy = UserNameNotTakenValidator.class)
@Documented
@Retention(RUNTIME)
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE})
public @interface UserNameNotTaken {

    String message() default "";

    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Класс `UserNameNotTakenValidator` проверяет это условие.

```
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class UserNameNotTakenValidator implements
ConstraintValidator<UserNameNotTaken, String> {

    @Inject
    UserStore userStore;

    public void initialize(UserNameNotTaken constraint) {
        // ничего не делать
    }

    public boolean isValid(String string, ConstraintValidatorContext context) {
        return !userStore.isNameTaken(string);
    }
}
```

Как и в случае с другими стандартами, валидаторы могут использовать управляемые объекты через внедрение зависимостей. Такая необходимость возникает очень часто при нестандартной логике валидации с вызовом элементов управления. В данном примере валидатор внедряет `UserStore`. Напомню еще раз: в Java EE различные стандарты можно применять многократно.

Специальные условия валидации очень часто определяются предметной областью. В виде специальных условий имеет смысл инкапсулировать сложную многокомпонентную логику валидации. При таком подходе эффективно используется также принцип единой ответственности — выделение логики проверки в единый валидатор вместо разделения ее на простейшие условия.

Bean Validation предоставляет более сложный функционал, позволяющий строить сценарии проверки, в которых к одним и тем же типам применяются различные способы валидации. Для этого реализуется концепция групп: условия объединяются в группы, которые могут быть проверены индивидуально. Подробнее об этом читайте в спецификации Bean Validation.

Как было показано ранее, полезные данные HTTP JSON также могут быть преобразованы в JAX-RS с использованием стандарта JSON-P. Это возможно и для тела запросов HTTP. Параметры тела запроса могут быть представлены как типы JSON-P, содержащие JSON-структуры, которые считываются динамически. Имеет смысл представлять тела запросов, как и тело отклика, с применением типов JSON-P, если структура объекта отличается от типов моделей или требует большей гибкости. В таких случаях выполнять валидацию предоставляемых объектов еще важнее, поскольку структуры JSON-P могут быть произвольными. Для того чтобы гарантировать правильность определенных свойств объекта запроса, представленных в формате JSON, этот объект должен пройти валидацию с использованием специальных условий проверки.

Поскольку объекты JSON-P построены программно и не имеют предопределенных типов, у программистов нет возможности аннотировать их поля, как это делается для типов Java. Поэтому к параметрам тела запроса применяются специальные условия валидации, привязанные к соответствующему валидатору. Специальные условия валидации определяют структуру приемлемого JSON-объекта для данного тела запроса.

В следующем примере представлен код для интеграции проверенного типа JSON-P в ресурсный метод JAX-RS:

```
@Path("users")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class UsersResource {

    …

    @POST
    public Response createUser(@Valid @ValidUser JsonObject json) {

        User user = readUser(json);
```

```
        long id = userStore.create(user);
        …
    }

    private User readUser(JsonObject object) {
        …
    }
}
```

Специальное условие `ValidUser` ссылается на используемый валидатор. Поскольку структура предоставляемых объектов JSON-P может быть произвольной, валидатор проверяет наличие и тип определенных свойств:

```
@Constraint(validatedBy = ValidUserValidator.class)
@Documented
@Retention(RUNTIME)
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE})
public @interface ValidUser {

    String message() default "";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

Валидатор специальных условий применим также к типам JSON-P:

```
public class ValidUserValidator implements ConstraintValidator<ValidUser,
JsonObject> {

    public void initialize(ValidUser constraint) {
        // ничего не делать
    }

    public boolean isValid(JsonObject json, ConstraintValidatorContext context) {
        …
    }
}
```

После того как предоставленный объект JSON-P прошел валидацию, его свойства можно безопасно извлечь. В этом примере показано, как гибкие программируемые типы можно интегрировать и валидировать с помощью методов JAX-RS. Класс ресурсов извлекает тело запроса, преобразует его в тип сущности предметной области и использует контур для вызова бизнес-сценария.

Обработка ошибок

Как мы видели в последних примерах, JAX-RS позволяет преобразовать исключения в специальные отклики. Эта полезная функциональность дает возможность реализовать прозрачную обработку специфических ошибок, не влияющую на выполнение основного кода.

Типичная проблема при работе с EJB состоит в том, что при получении доступа к любому контексту, не являющемуся EJB-объектом, например запросу на ресурс JAX-RS, все исключения будут обернуты в `EJBException`. Это делает обработку исключений очень громоздкой, потому что `EJBException` должно быть развернуто для проверки причины исключения.

Если снабжать специальные типы исключений аннотациями `@ApplicationException`, то причина исключения не будет обернута:

```
import javax.ejb.ApplicationException;

@ApplicationException
public class GreetingException extends RuntimeException {
    public GreetingException(String message) {
        super(message);
    }
}
```

Вызов EJB, генерирующий исключение `GreetingException`, не приведет к появлению обернутого исключения `EJBException` и не создает тип исключения напрямую. Затем приложение может определить механизм преобразования исключений JAX-RS для фактического типа исключения `GreetingException`, так же как в случае преобразования условий валидации.

Благодаря условию аннотации `@ApplicationException (rollback = true)` контейнер также выполнит откат активной транзакции при возникновении исключения.

Доступ к внешним системам

Итак, теперь мы знаем, как обеспечивается доступ к бизнес-сценариям извне через HTTP.

Для реализации бизнес-логики большинству корпоративных приложений нужен доступ и к другим внешним системам (кроме баз данных, которые принадлежат приложению). Обычно под внешними системами подразумеваются системы, внешние по отношению к предметной области приложения. Они принадлежат другому контексту.

Для того чтобы получить доступ к внешним сервисам HTTP, нужно интегрировать в проект клиентский компонент, как правило, в виде отдельного элемента управления. Этот класс элемента управления инкапсулирует функциональные возможности, необходимые для связи с внешней системой. Рекомендуется тщательно проработать интерфейс и не смешивать задачи предметной области с деталями реализации соединения, такими как упаковка потенциальных полезных данных, протокол связи, информация HTTP (если используется HTTP) и другие составляющие, не относящиеся к основной предметной области.

В состав JAX-RS входит сложная клиентская функция для обеспечения эффективного доступа к HTTP-сервисам. Она предоставляет такие же возможности преобразования типов, что и для классов ресурсов. В следующем примере

представлен элемент управления для доступа к внешней системе для подачи кофейных зерен:

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.enterprise.context.ApplicationScoped;
import javax.ws.rs.client.*;
import java.util.concurrent.TimeUnit;

@ApplicationScoped
public class CoffeePurchaser {

    private Client client;
    private WebTarget target;

    @PostConstruct
    private void initClient() {
        client = ClientBuilder.newClient();
        target = client.target("http://coffee.example.com/beans/purchases/");
    }

    public OrderId purchaseBeans(BeanType type) {
        Purchase purchase = …

        BeanOrder beanOrder = target
                .request(MediaType.APPLICATION_JSON_TYPE)
                .post(Entity.json(purchase))
                .readEntity(BeanOrder.class);

        return beanOrder.getId();
    }

    @PreDestroy
    public void closeClient() {
        client.close();
    }
}
```

Клиент JAX-RS создается и настраивается клиентом-строителем. Он задействует целевые веб-объекты для доступа к URL. Эти объекты могут быть изменены с использованием функций построения URI, аналогичных тем, которые действуют в ресурсах JAX-RS. Целевые объекты применяются для создания новых вызовов, представляющих фактические HTTP-вызовы. Вызовы могут настраиваться в соответствии с HTTP-информацией, такой как типы содержимого, заголовки, а также особенности преобразуемых типов Java.

В данном примере целевой объект, указывающий на внешний URL, создает новый запрос содержимого типа JSON с помощью метода HTTP POST. Ожидается, что возвращаемая JSON-структура будет преобразовываться в объект `BeanOrder`. Дальнейшую логику для извлечения необходимой информации выполняет клиент.

Во избежание утечки ресурсов при завершении работы контейнера в методе `@PreDestroy` экземпляр клиента будет закрыт должным образом.

Стабильность при использовании HTTP

В рассмотренном примере не учтены характеристики устойчивости. Вызов этого клиентского элемента управления без специальной проработки может привести к нежелательному поведению.

Запрос клиента блокируется до тех пор, пока на HTTP-вызов не будет получен ответ либо не истечет время обработки соединения. Допустимое время обработки HTTP-соединения зависит от реализации JAX-RS, которая в отдельных технологиях настроена на бесконечную блокировку. Для стабильных клиентов это неприемлемо. Соединение может длиться вечно, блокируя поток, и в худшем случае может заблокировать все приложение, если все доступные потоки будут привязаны к этой локации, ожидая завершения их индивидуального HTTP-соединения. Чтобы это предотвратить, нужно настроить клиент, задав для него допустимое время ожидания соединения.

Время ожидания зависит от приложения, особенно от параметров сети, по которой осуществляется связь с внешней системой. Оно определяется допустимыми значениями времени ожидания HTTP. Для того чтобы получить разумное время ожидания, рекомендуется собирать статистику задержек доступа к внешней системе. Для систем, в которых нагрузка и время ожидания соединения сильно различаются, например для систем электронной коммерции с пиковой нагрузкой в определенное время, следует учитывать характер изменений.

Время ожидания HTTP-соединения — это максимальное время, в течение которого соединение должно быть установлено. Оно должно быть небольшим. Время ожидания чтения HTTP определяет, как долго можно ожидать чтения данных. Оно зависит от характера внешнего сервиса. В соответствии с собранной статистикой хорошей отправной точкой для выбора времени ожидания чтения является среднее время отклика плюс трехкратное стандартное отклонение. Подробнее темы производительности и реакции сервисов будут рассмотрены в главе 9.

В следующем примере показано, как настроить HTTP-соединение и время ожидания чтения:

```
@ApplicationScoped
public class CoffeePurchaser {

    …

    @PostConstruct
    private void initClient() {
        client = ClientBuilder.newBuilder()
                .connectTimeout(100, TimeUnit.MILLISECONDS)
                .readTimeout(2, TimeUnit.SECONDS)
                .build();
        target = client.target("http://coffee.example.com/beans/purchases/");
    }

    …
}
```

Вызовы клиентов могут привести к появлению ошибок. Внешняя служба может выдать непредвиденный код состояния, дать неожиданный ответ или не ответить вообще. Это необходимо учитывать при реализации клиентских компонентов.

Клиентский вызов `readResponse()` ожидает, что ответом будет код состояния HTTP типа `SUCCESSFUL`, а тело ответа можно будет преобразовать из типа содержимого запроса в заданный тип Java. Если что-то пойдет не так, будет сгенерировано динамическое исключение `RuntimeException`. Динамические исключения позволяют инженерам писать код, не засоренный лишними блоками обработки исключений, но при этом необходимо учитывать возможные ошибки.

Клиентский метод может улавливать динамические исключения, чтобы они не передавались вызывающему сервису предметной области. Существует и другой, более простой вариант: использовать перехватчики. Перехватчики — это сквозные функции, которые применяются без жесткой привязки к декорированным функциям. Например, рассмотренный клиентский метод должен возвращать значение `null`, если внешняя система не предоставила приемлемый ответ.

В следующем примере показан перехватчик, который перехватывает вызовы методов и ведет себя так по отношению к сгенерированным исключениям. Данный перехватчик интегрируется с помощью аннотации метода управления `CoffeePurchaser`:

```
import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

@Interceptor
public class FailureToNullInterceptor {

    @AroundInvoke
    public Object aroundInvoke(InvocationContext context) {
        try {
            return context.proceed();
        } catch (Exception e) {
            …
            return null;
        }
    }
}
```

Метод `purchaseBean()` снабжен аннотацией `@Interceptors(FailureToNullInterceptor.class)`, которая активизирует для него сквозные функции.

Что касается устойчивости, то функции клиента могут включать в себя дополнительную логику. Если доступны несколько систем, то в случае неудачи клиент может повторить вызов, обратившись к другой системе. И только в крайнем случае вызов закончится неудачей и результат не будет получен.

Подробнее сквозные функции будут рассмотрены в разделе «Сквозные задачи» далее.

Доступ к сервисам REST Hypermedia

HTTP-сервисы, использующие условия REST, особенно в отношении Hypermedia, нуждаются в более сложной логике на стороне клиента. Сервисы направляют клиентов на соответствующие ресурсы, доступ к которым осуществляется определенным образом. Hypermedia разделяет сервисы и применяет функции API, такие как способность к развитию и открытость, но требует большей динамичности и логики на стороне клиента.

Ранее был представлен пример для типа содержимого Siren, в котором было показано, как отклик сервиса перенаправляет REST-клиента на доступные последующие вызовы. Предположим, что в ответе клиент получает заказ и хочет выполнить действие `add-to-cart`:

```
{
    ... рассмотренный ранее пример
    ... свойства ресурса book
    "actions": [
        {
            "name": "add-to-cart",
            "title": "Add Book to cart",
            "method": "POST",
            "href": "http://api.example.com/shopping-cart",
            "type": "application/json",
            "fields": [
                { "name": "isbn", "type": "text" },
                { "name": "quantity", "type": "number" }
            ]
        }
    ],
    "links": …
}
```

Клиент знает только то, что означает действие «добавить в корзину» с точки зрения бизнес-сценария, а также как предоставить сведения о значении полей ISBN и количестве книг. Эта логика предметной области, безусловно, должна быть реализована в клиенте. Информация о том, как осуществляется доступ к следующему ресурсу — корзине покупок с использованием метода HTTP и каким будет тип содержимого, является динамической и не записана в клиенте.

Для того чтобы добавить книгу в корзину покупок, клиент сначала должен получить доступ к ресурсу книги. Затем вызывается бизнес-сценарий «добавить в корзину», он извлекает информацию о соответствующем действии Hypermedia. Информация для нужных полей должна быть предоставлена при вызове. После этого клиент обращается ко второму ресурсу, используя сведения, предоставленные REST-сервисом и вызовом элемента управления:

```
public class BookClient {

    @Inject
    EntityMapper entityMapper;

    public Book retrieveBook(URI uri) {
```

```
        Entity book = retrieveEntity(uri);
        return entityMapper.decodeBook(uri, book.getProperties());
    }

    public void addToCart(Book book, int quantity) {
        Entity bookEntity = retrieveEntity(book.getUri());

        JsonObjectBuilder properties = Json.createObjectBuilder();
        properties.add("quantity", quantity);

        Entity entity = entityMapper.encodeBook(book);
        entity.getProperties().forEach(properties::add);

        performAction(bookEntity, "add-to-cart", properties.build());
    }

    private Entity retrieveEntity(URI uri) {
        …
    }

    private void performAction(Entity entity, String actionName,
            JsonObject properties) {
        …
    }
}
```

Тип `Entity` инкапсулирует информацию типов сущностей Hypermedia. `EntityMapper` отвечает за преобразование типа содержимого в модели предметной области и наоборот. В этом примере все необходимые поля для действия получены из свойств ресурса и предоставленного параметра `quantity`. Для того чтобы активизировать определенную динамику, все свойства сущности добавляются в карту преобразования и передаются методу `performAction()`. В зависимости от действия, заданного сервером, из этой карты извлекаются необходимые поля. Если требуется больше полей, то логику клиента, очевидно, следует изменить.

Разумеется, имеет смысл инкапсулировать логику доступа к сервисам Hypermedia, а также преобразование моделей предметной области в типы содержимого, отдельные делегаты. Также имеет смысл заменить библиотекой функциональность для доступа к REST-сервисам.

Вы могли заметить, что URI просочились в открытый интерфейс класса клиента. И не случайно — это необходимо, чтобы идентифицировать ресурсы для нескольких вызовов бизнес-сценариев. Другими словами, URI перешли в предметную область и используются там как универсальные идентификаторы ресурсов. Поскольку логика создания URL на основе технических идентификаторов выполняется на стороне клиента, весь URL-адрес ресурса, в сущности, становится *идентификатором*. Однако при разработке клиентских элементов управления инженерам следует позаботиться об открытом интерфейсе. В частности, никакая информация о связи с внешней системой не должна попасть в предметную область. Этот подход хорошо реализуется с помощью Hypermedia. Вся необходимая транспортная информация извлекается и используется динамически. Логика

навигации, реализуемая в соответствии с ответами Hypermedia, находится в клиентском элементе управления.

Этот пример призван дать читателю представление о том, как клиент задействует сервисы REST Hypermedia.

Асинхронная коммуникация и обмен сообщениями

Асинхронная коммуникация приводит к ослаблению связи между системами. Обычно это повышает оперативность, а также увеличивает накладные расходы и позволяет выполнять сценарии, в которых отсутствует постоянная надежная связь с системами. Есть множество способов проектирования асинхронных соединений на концептуальном и техническом уровне. Асинхронная связь не означает, что на техническом уровне вызовы никогда не будут синхронными. Бизнес-процесс может быть построен как асинхронный, но при этом моделировать один или несколько синхронных вызовов, вот только они не будут выполняться или обрабатываться немедленно. Например, API может предусматривать синхронные методы для создания долговременных процессов, которые впоследствии будут часто опрашиваться на предмет обновлений.

На техническом уровне асинхронная связь обычно разрабатывается как ориентированная на сообщения и реализуется на основе очередей сообщений или шаблона публикации-подписки. Приложения общаются напрямую только с очередью или брокером сообщений, а сообщения не передаются напрямую конкретному получателю.

Рассмотрим различные способы создания асинхронных соединений.

Асинхронное HTTP-соединение

Модель ответа на запрос при HTTP-соединении обычно включает в себя синхронный обмен данными. Клиент запрашивает ресурс на сервере и блокируется до тех пор, пока не будет передан ответ. Таким образом, асинхронная связь с использованием HTTP обычно достигается на концептуальной основе. Синхронные HTTP-вызовы могут инициировать длительные бизнес-процессы. Затем внешняя система может либо уведомить вызывающего абонента посредством другого механизма, либо предложить вариант опроса для обновлений.

Например, в сложной системе управления пользователями предусмотрены методы для создания пользователей. Допустим, пользователей требуется регистрировать и верифицировать во внешних системах и этот шаг входит в более длительный асинхронный бизнес-процесс. Затем приложение с помощью HTTP-функции, например `POST /users/`, запускает процесс создания новых пользователей. Однако вызов такого сценария не гарантирует, что пользователь будет успешно создан и зарегистрирован. Ответ от конечной точки HTTP, например, с кодом статуса `202 Accepted` будет только подтверждать попытку создания нового пользователя. Он говорит лишь о том, что запрос принят, но это еще не значит, что он был полностью обработан. Дальше с помощью поля заголовка `Location` можно перенаправить клиента на ресурс, где он сможет выполнять опрос обновлений для частично зарегистрированного пользователя.

На техническом уровне HTTP не только поддерживает синхронные вызовы. В пункте «События, посылаемые сервером» мы рассмотрим отправляемые с сервера события как пример HTTP-стандарта, использующего асинхронную связь посредством сообщений.

Связь посредством сообщений

Связь посредством сообщений подразумевает обмен информацией, содержащейся в асинхронно отправляемых сообщениях. Обычно она реализуется с помощью очередей сообщений или шаблона публикации-подписки. Преимуществом такой связи является разделение систем: приложения напрямую взаимодействуют только с очередью или брокером сообщений. Разделение влияет не только на зависимость от систем и используемой технологии, но и на характер коммуникации, так как благодаря асинхронному обмену сообщениями бизнес-процессы становятся слабее связанными между собой.

Очередь сообщений — это очередь, в которую помещаются сообщения, чтобы затем по одному быть переданными потребителям. В корпоративных системах очереди сообщений обычно реализуются посредством *промежуточного ПО, ориентированного на обработку сообщений (message-oriented middleware, MOM)*. В прошлом решения MOM часто встречались в системах очередей сообщений, таких как ActiveMQ, RabbitMQ и WebSphere MQ.

Шаблон публикации-подписки описывает потребителей, которые подписываются на определенную тему и получают публикуемые в ней сообщения. Подписчики регистрируются в теме и получают сообщения, отправленные издателем. Эта концепция хорошо масштабируется для большого числа участников. Промежуточное программное обеспечение, ориентированное на обработку сообщений, обычно применяют, чтобы использовать преимущества как очередей сообщений, так и шаблона публикации-подписки.

Однако, как и асинхронная связь в целом, программные решения, ориентированные на сообщения, имеют определенные недостатки. Первое, о чем следует позаботиться, — это надежная доставка сообщений. Издатели отправляют сообщения в асинхронном, *автономном* режиме. Инженеры должны учитывать поддерживаемую семантику доставки сообщений: будет ли сообщение получено *максимум один раз, хотя бы один раз* или *ровно один раз*. Выбор технологии, поддерживающей определенную семантику доставки, особенно вариант доставки ровно один раз, будет влиять на масштабируемость и пропускную способность системы. В главе 8 мы подробно рассмотрим эту тему, когда будем обсуждать приложения, управляемые событиями.

Для интеграции в приложения Java EE промежуточного программного обеспечения, ориентированного на обработку сообщений, можно использовать API *Java Message Service (JMS)*. JMS API поддерживает решения и для очередей сообщений, и для шаблонов публикации-подписки. Этот API только определяет интерфейсы и реализуется с помощью промежуточного ПО, ориентированного на обработку сообщений. Несмотря на это JMS API не получил широкого распространения среди программистов. На момент написания книги существовало не так уж много систем, в которых он применялся. По

сравнению с другими стандартами его модель программирования сложна и малопродуктивна.

Еще одна тенденция в коммуникации посредством сообщений заключается в том, что на смену традиционным MOM-решениям все чаще приходят более простые. Сейчас многие из решений, ориентированных на сообщения, интегрируются с помощью собственных API. Примером является Apache Kafka, в котором используется и очередь сообщений, и модель публикации-подписки. В главе 8 в качестве примера применения MOM-решения в приложениях Java EE показана интеграция Apache Kafka.

События, посылаемые сервером

События, посылаемые сервером (server-sent events, SSE), — пример асинхронной технологии публикации-подписки на основе HTTP. Она предлагает простой протокол однонаправленной потоковой передачи. Клиенты могут зарегистрироваться в теме, запросив HTTP-ресурс, который оставляет открытое соединение. Сервер посылает сообщения клиентам, подключенным к нему по этим активным HTTP-соединениям. Клиенты не могут ответить ему напрямую, а могут только открывать и закрывать соединения с конечной точкой потоковой передачи. Это простое решение подходит для сценариев с новостными рассылками, такими как обновления в социальных сетях, рассылка цен на акции или новостные ленты.

Сервер рассылает клиентам, подписанным на тему, текстовые данные в формате UTF-8 в виде содержимого `text/event-stream`. Формат событий показан далее:

```
data: Это сообщение

event: namedmessage
data: У этого сообщения есть имя события

id: 10
data: У этого сообщения есть id; если соединение будет прервано, он будет передан
как 'ID последнего события'
```

Тот факт, что технология событий, отправляемых сервером, основана на HTTP, упрощает ее интеграцию в существующие сети и инструментарий разработки. SSE изначально поддерживает идентификаторы событий и восстановление соединений. Клиенты, которые повторно подключаются к конечной точке потоковой передачи, предоставляют последний полученный идентификатор события и продолжают подписку с того места, на котором остановились.

JAX-RS поддерживает события, посылаемые сервером, как на стороне сервера, так и на стороне клиента. Конечные точки передачи SSE определяются с использованием ресурсов JAX-RS следующим образом:

```
import javax.ws.rs.DefaultValue;
import javax.ws.rs.HeaderParam;
import javax.ws.rs.InternalServerErrorException;
import javax.ws.rs.core.HttpHeaders;
```

```
import javax.ws.rs.sse.*;

@Path("events-examples")
@Singleton
public class EventsResource {

    @Context
    Sse sse;

    private SseBroadcaster sseBroadcaster;
    private int lastEventId;
    private List<String> messages = new ArrayList<>();

    @PostConstruct
    public void initSse() {
        sseBroadcaster = sse.newBroadcaster();

        sseBroadcaster.onError((o, e) -> {
            …
        });
    }

    @GET
    @Lock(READ)
    @Produces(MediaType.SERVER_SENT_EVENTS)
    public void itemEvents(@HeaderParam(HttpHeaders.LAST_EVENT_ID_HEADER)
                           @DefaultValue("-1") int lastEventId,
                           @Context SseEventSink eventSink) {

        if (lastEventId >= 0)
            replayLastMessages(lastEventId, eventSink);

        sseBroadcaster.register(eventSink);
    }

    private void replayLastMessages(int lastEventId, SseEventSink eventSink) {
        try {
            for (int i = lastEventId; i < messages.size(); i++) {
                eventSink.send(createEvent(messages.get(i), i + 1));
            }
        } catch (Exception e) {
            throw new InternalServerErrorException("Не удается повторить
                      сообщения", e);
        }
    }

    private OutboundSseEvent createEvent(String message, int id) {
        return
            sse.newEventBuilder().id(String.valueOf(id)).data(message).build();
    }

    @Lock(WRITE)
    public void onEvent(@Observes DomainEvent domainEvent) {
        String message = domainEvent.getContents();
```

```
        messages.add(message);

        OutboundSseEvent event = createEvent(message, ++lastEventId);

        sseBroadcaster.broadcast(event);
    }
}
```

Для событий, посылаемых сервером, используется тип содержимого `text/event-stream`. Зарегистрированный приемник событий `SseEventSink` дает JAX-RS инструкцию поддерживать открытое соединение с клиентом для следующих событий, которые будут отправлены через передатчик. Согласно стандарту SSE заголовок `Last-Event-ID` управляет продолжением потока событий. В данном примере сервер будет повторно отправлять сообщения, если они публиковались, пока клиенты были отключены.

В методе `itemEvents()` реализована регистрация потоковой передачи и немедленной повторной отправки пропущенных событий клиенту в случае необходимости. После того как клиент зарегистрировал результат, он вместе с другими активными клиентами продолжит получать следующие сообщения, созданные с помощью `Sse`.

В корпоративном приложении асинхронная интеграция реализуется через отслеживание события `DomainEvent`. Каждый раз, когда где-то в приложении происходит CDI-событие этого типа, активные клиенты SSE получают сообщение.

JAX-RS также поддерживает возможность использования SSE. Источник событий `SseEventSource` предоставляет функциональные возможности для открытия соединения с конечной точкой SSE. Он регистрирует получатель событий, который вызывается при поступлении сообщений:

```
import java.util.function.Consumer;

public class SseClient {

    private final WebTarget target =
        ClientBuilder.newClient().target("...");
    private SseEventSource eventSource;

    public void connect(Consumer<String> dataConsumer) {
        eventSource = SseEventSource.target(target).build();

        eventSource.register(
                item -> dataConsumer.accept(item.readData()),
                Throwable::printStackTrace,
                () -> System.out.println("completed"));

        eventSource.open();
    }

    public void disconnect() {
        if (eventSource != null)
            eventSource.close();
    }
}
```

После того как `SseEventSource` успешно откроет соединение, текущий поток продолжается. Когда события поступят, будет вызван приемник событий, в данном случае `dataConsumer#accept`. `SseEventSource` станет выполнять всю необходимую обработку в соответствии со стандартом SSE. Сюда входят, в частности, повторное подключение после разрыва соединения и отправка заголовка `Last-Event-ID`.

У клиентов также есть возможность построения более сложных решений с ручным управлением заголовками и повторными подключениями. Поэтому тип `SseEventInput` запрашивается с типом содержимого `text/event-stream` из обычного целевого веб-объекта. Подробнее об этом читайте в спецификации JAX-RS.

События, посылаемые сервером, предоставляют удобное одностороннее потоковое решение для HTTP, которое хорошо интегрируется в технологию Java EE.

WebSocket

Технология событий, посылаемых сервером, конкурирует с более мощной технологией *WebSocket*, которая поддерживает двустороннюю связь. Технология WebSocket, стандартизованная IETF, является еще одним примером ориентированной на сообщения связи типа «публикация-подписка». Изначально она предназначалась для приложений на основе браузера, но может применяться для любого клиент-серверного обмена сообщениями. В WebSocket обычно используются те же порты, что и на конечных точках HTTP, но с собственным TCP-протоколом.

В Java EE WebSocket является частью интерфейса *Java API for WebSocket*, включающего в себя сервер и поддержку клиент-серверного обмена данными.

Модель программирования для определения конечных точек на стороне сервера здесь тоже совпадает с общей картиной, характерной для Java EE. Конечные точки могут быть определены программно или декларативно, с помощью аннотаций. Последний вариант предполагает, что аннотации добавляются в классы конечных точек, как в модели программирования ресурсов JAX-RS:

```
import javax.websocket.*;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint(value = "/chat", decoders = ChatMessageDecoder.class,
encoders = ChatMessageEncoder.class)
public class ChatServer {

    @Inject
    ChatHandler chatHandler;

    @OnOpen
    public void openSession(Session session) {
        …
    }

    @OnMessage
    public void onMessage(ChatMessage message, Session session) {
```

```
        chatHandler.store(message);
    }

    @OnClose
    public void closeSession(Session session) {
        …
    }
}
```

Аннотированные методы класса конечной точки сервера будут вызываться в инициирующих сессиях, при поступлении сообщений и закрытии соединений. Сессии представляют собой обмен данными между двумя конечными точками.

Конечными точками WebSocket могут определяться кодеры и декодеры для преобразования пользовательских типов Java из текста в двоичные данные и наоборот. В этом примере представлен нестандартный тип для сообщений чата, для преобразования которого используются специальные кодеры и декодеры. Подобно JAX-RS, WebSocket поставляется с готовыми средствами сериализации для обычных сериализуемых типов Java, таких как строки. В следующем коде продемонстрирован кодер для специального типа, созданного для нашей предметной области:

```
import javax.websocket.EncodeException;
import javax.websocket.Encoder;
import javax.websocket.EndpointConfig;

public class ChatMessageEncoder implements Encoder.Binary<ChatMessage> {

    @Override
    public ByteBuffer encode(ChatMessage object) throws EncodeException {
        …
    }

    …
}
```

Эти типы соответствуют типам `MessageBodyWriter` и `MessageBodyReader` в стандарте JAX-RS. Далее показан соответствующий декодер сообщений:

```
import javax.websocket.DecodeException;
import javax.websocket.Decoder;
import javax.websocket.EndpointConfig;

public class ChatMessageDecoder implements Decoder.Binary<ChatMessage> {

    @Override
    public ChatMessage decode(ByteBuffer bytes) throws DecodeException {
        …
    }

    …
}
```

Конечные точки клиента определяются аналогично конечным точкам сервера. Разница лишь в том, что серверы WebSocket прослушивают новое соединение вдоль всего пути, а клиенты — нет.

Клиентские функциональные возможности WebSocket API могут использоваться не только в корпоративной среде, но и в приложениях Java SE. То же самое касается и JAX-RS на стороне клиента. Реализацию конечной точки клиента WebSocket предоставлю вам в качестве упражнения.

WebSocket и события, посылаемые сервером, представляют собой хорошо интегрированные технологии, ориентированные на сообщения. Какую из них применять, в значительной степени зависит от требований предметной области, существующего окружения и характера обмена данными.

Корпоративные технологии связи

Некоторые внешние корпоративные системы, которые нужно интегрировать с приложением, не имеют стандартных интерфейсов или Java API. К этой категории могут относиться устаревшие системы, а также другие системы, используемые в организации. API *Java EE Connector Architecture (JCA)* позволяет интегрировать эти так называемые *корпоративные информационные системы (Enterprise Information Systems, EIS)* в приложения Java EE. Примерами EIS являются системы обработки транзакций, системы обмена сообщениями и закрытые базы данных.

Адаптеры ресурсов JCA представляют собой развертываемые компоненты EE, которые позволяют интегрировать в приложение информационные системы: соединения, транзакции, безопасность и управление жизненным циклом. Информационные системы могут лучше интегрироваться в приложение, чем другие технологии подключения. Адаптеры ресурсов поставляются в виде *архивов адаптеров ресурсов (Resource Adapter Archives, RAR)*. Доступ к ним в приложении осуществляется благодаря функционалу пакета `javax.resource` и его подпакетов. Некоторые поставщики EIS предоставляют адаптеры ресурсов для своих систем. Подробнее о разработке и развертывании адаптеров ресурсов читайте в спецификации JCA.

JCA предлагает множество вариантов интеграции для внешних информационных систем. Однако этот стандарт не получил широкого применения и не пользуется признанием у инженеров корпоративных систем. Разработка адаптеров ресурсов — довольно сложное дело, а интерфейс JCA API не очень хорошо известен программистам, поэтому компании обычно предпочитают интегрировать системы другими способами. На практике следует решить, что лучше: потратить силы на то, чтобы написать адаптер ресурсов, или интегрировать информационные системы с помощью других технологий. Под другими решениями здесь подразумеваются такие интеграционные структуры, как Apache Camel и Mule ESB.

Системы управления базами данных

Большинство корпоративных приложений задействуют базы данных в качестве постоянного хранилища данных. Базы данных лежат в основе корпоративной системы, в которой содержатся данные приложения. Сейчас данные являются одним из самых важных товаров. Компании тратят много времени и усилий на их сбор, защиту и применение.

Существует несколько способов представления состояния в корпоративных системах, однако реляционные базы данных по-прежнему остаются самыми популярными. Концепции и способы их использования глубоко изучены и хорошо интегрированы в корпоративные технологии.

Интегрирование систем реляционных баз данных

Java Persistence API (JPA) применяется для интеграции реляционных систем баз данных в корпоративные приложения. JPA, если сравнивать с устаревшими подходами времен J2EE, хорошо интегрируется с моделями предметной области, построенными в соответствии с концепциями проблемно-ориентированного проектирования. Постоянное хранение сущностей не требует больших расходов и не накладывает особых ограничений на модель. Это позволяет сначала построить модель предметной области, сосредоточившись на бизнес-аспектах, а затем интегрировать уровень базы данных.

Взаимодействие с базой данных интегрируется с предметной областью как неотъемлемая часть бизнес-сценария. В зависимости от сложности последнего функции взаимодействия с базой данных вызываются либо через специальные элементы управления, либо непосредственно из контура. Проблемно-ориентированное проектирование предусматривает концепцию репозиториев, которая, как уже отмечалось, хорошо соответствует обязанностям диспетчера сущностей JPA. Диспетчер сущностей используется для получения и сохранения объектов, управления ими, а также выполнения запросов. Его интерфейс является абстрактным, и его можно задействовать в обобщенной форме.

Во времена J2EE применялся шаблон *объекта доступа к данным (Data Access Object, DAO)*. Причиной этого было абстрагирование и инкапсулирование функций доступа к данным. Этот шаблон определяет тип доступной системы хранения, такой как реляционные базы данных, объектно-ориентированные базы данных, системы LDAP или файлы. Такой подход, конечно, имеет смысл, но с появлением Java EE необходимость в нем практически отпала.

В большинстве корпоративных приложений используются реляционные базы данных, поддерживающие SQL и JDBC. В JPA уже реализована абстракция систем реляционных баз данных, так что инженерам обычно не приходится иметь дело с особенностями поставщика. Однако применение вместо СУРБД другой системы хранения повлияет на код приложения. Преобразование типов сущностей предметной области в объекты хранения больше не требует задействования объектов переноса, поскольку JPA хорошо интегрируется с моделями предметной области. Непосредственное преобразование типов сущностей домена — это эффективный способ реализовать взаимодействие с базой данных без особых накладных расходов. Таким образом, в простых случаях, например при сохранении и чтении объектов, подход DAO применять не требуется. Однако для сложных запросов к базе данных имеет смысл инкапсулировать эту функциональность в отдельных элементах управления. Такие репозитории могут хранить все функ-

ции взаимодействия с базой данных для конкретных типов сущностей. Однако рекомендуется начать с простого подхода и только по мере возрастания сложности выделять взаимодействие с базой данных в особую точку ответственности.

Таким образом, для управления сохранением сущностей контуры и элементы управления используют диспетчер сущностей. В следующем примере показано, как интегрировать диспетчер сущности в контур:

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class PersonAdministration {
    @PersistenceContext
    EntityManager entityManager;

    public void createPerson(Person person) {
        entityManager.persist(person);
    }

    public void updateAddress(long personId, Address newAddress) {
        Person person = entityManager.find(Person.class, personId);

        if (person == null)
            throw new IllegalArgumentException("Не удалось найти человека с ID "
                      + personId);

        person.setAddress(newAddress);
    }
}
```

Операция `persist()` при создании новых объектов `Person` делает `person` управляемой сущностью. Она будет добавлена в базу данных после завершения транзакции и может быть извлечена оттуда впоследствии по присвоенному ей идентификатору, как сделано в методе `updateAddress()`. Сущность `person` извлекается из базы по ID и преобразуется в управляемую сущность. Все изменения сущности будут синхронизироваться с базой данных в момент завершения транзакции.

Преобразование моделей предметной области

Как уже отмечалось, сущности, агрегаты и объекты-значения интегрированы в JPA без особых ограничений модели. Сущности, как и агрегаты, представлены в виде сущностей JPA:

```
import javax.persistence.*;

@Entity
@Table(name = "persons")
public class Person {

    @Id
    @GeneratedValue
    private long id;
```

```
    @Basic(optional = false)
    private String name;

    @Embedded
    private Address address;
    …
}

@Embeddable
public class Address {

    @Basic(optional = false)
    private String streetName;

    @Basic(optional = false)
    private String postalCode;

    @Basic(optional = false)
    private String city;

    …
}
```

Тип `person` — это сущность. Его необходимо идентифицировать с помощью ID, который будет первичным ключом в таблице `persons`. Каждое свойство записывается в базу данных определенным образом в зависимости от характера типа и отношения. Например, имя человека — это простой текстовый столбец.

Адрес — это неидентифицируемый объект-значение. С точки зрения предметной области не важно, *к какому именно адресу* мы обращаемся, лишь бы совпадали значения. Поэтому адрес не является сущностью и не отображается в JPA. Объекты-значения могут быть реализованы с помощью встраиваемых типов JPA. Свойства таких типов будут сохраняться в базе данных в виде дополнительных столбцов таблицы сущности, содержащих ссылки на них. Поскольку объект `person` включает в себя определенное значение адреса, свойства адреса будут частью таблицы `person`.

Корневые агрегаты, состоящие из нескольких сущностей, могут быть реализованы в виде отношений, сохраняемых в соответствующих столбцах и таблицах базы данных. Например, автомобиль состоит из двигателя, одного или нескольких сидений, кузова и многих других деталей. Некоторые из них являются сущностями, которые потенциально могут быть идентифицированы и доступны как отдельные объекты. Производитель автомобиля может идентифицировать весь автомобиль или только двигатель и в случае ремонта или замены действовать соответственно. Представление базы данных также можно разместить поверх существующей модели предметной области.

Далее приводятся фрагменты кода с описанием сущности автомобиля, включая отображение JPA:

```
import javax.persistence.CascadeType;
import javax.persistence.OneToMany;
import javax.persistence.OneToOne;
```

```
@Entity
@Table(name = "cars")
public class Car {

    @Id
    @GeneratedValue
    private long id;

    @OneToOne(optional = false, cascade = CascadeType.ALL)
    private Engine engine;

    @OneToMany(cascade = CascadeType.ALL)
    private Set<Seat> seats = new HashSet<>();

    …
}
```

Сиденья включены в коллекцию `HashSet`, которая создается для новых экземпляров `Car`. Следует избегать коллекций Java со значением `null`.

Двигатель представляет собой другую сущность предметной области:

```
import javax.persistence.EnumType;
import javax.persistence.Enumerated;

@Entity
@Table(name = "engines")
public class Engine {

    @Id
    @GeneratedValue
    private long id;

    @Basic(optional = false)
    @Enumerated(EnumType.STRING)
    private EngineType type;

    private double ccm;

    …
}
```

Сиденья также являются сущностями, идентифицируемыми по ID:

```
@Entity
@Table(name = "seats")
public class Seat {

    @Id
    @GeneratedValue
    private long id;

    @Basic(optional = false)
    @Enumerated(EnumType.STRING)
    private SeatMaterial material;
```

```
    @Basic(optional = false)
    @Enumerated(EnumType.STRING)
    private SeatShape shape;

    …
}
```

Все сущности, как автономные, так и зависимые от других объектов, должны быть управляемыми в контексте хранения в базе данных. Если двигатель автомобиля заменить новой сущностью, ее необходимо сохранить в базе отдельно. Операции сохранения должны вызываться либо явно, отдельными сущностями, либо каскадно по иерархии объектов. Каскады описываются в отношениях сущности. В следующем коде продемонстрированы оба подхода к сохранению данных о новом двигателе автомобиля:

```
public void replaceEngine(long carIdentifier, Engine engine) {
    Car car = entityManager.find(Car.class, carIdentifier);
    car.replaceEngine(engine);

    // автомобиль – управляемая сущность, двигатель нужно заменить
    entityManager.persist(engine);
}
```

После загрузки из базы данных информации об автомобиле по его идентификатору получаем управляемую сущность. Двигатель все еще нужно сохранить. Первый вариант — явное сохранение в виде сервиса.

Второй вариант — каскадная операция слияния, которая также обрабатывает новые сущности из составного объекта «автомобиль»:

```
public void replaceEngine(long carIdentifier, Engine engine) {
    Car car = entityManager.find(Car.class, carIdentifier);
    car.replaceEngine(engine);

    // операция слияния применяется к автомобилю и всем каскадным отношениям
    entityManager.merge(car);
}
```

Настоятельно рекомендую именно последний подход. Корневые составные объекты отвечают за поддержание целостного и согласованного общего состояния. Целостность данных проще гарантировать, если все операции инициируются и каскадируются из такого составного объекта.

Интеграция систем баз данных

Диспетчер сущностей управляет сущностями в контексте сохранения в базе данных. Он использует общий блок управления данными, соответствующий экземпляру базы данных. Блоки управления данными включают в себя все управляемые сущности, диспетчеры сущностей и преобразование конфигураций. При доступе только к одному экземпляру базы данных доступ к диспетчеру

сущностей может осуществляться напрямую, как было показано в предыдущем примере. Аннотация контекста сохранения в этом случае относится к одному блоку управления данными.

Блоки управления данными определены в дескрипторном файле `persistence.xml`, который находится в каталоге `META-INF`. Это один из немногих случаев в современной Java EE, когда действует конфигурация на основе XML. Дескриптор управления базой данных содержит блок управления данными и дополнительную конфигурацию. К источнику данных обращаются только по его JNDI-имени, чтобы отделить конфигурацию для доступа к экземпляру базы данных от приложения. Действительная конфигурация источника данных описана на сервере приложений. Если сервер приложений содержит только одно приложение, использующее одну базу данных, программисты могут задействовать источник данных сервера приложений по умолчанию. В этом случае имя источника данных можно пропустить.

В следующем фрагменте кода представлен пример файла `persistence.xml`, где описан единый блок управления данными, использующий источник данных по умолчанию:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2"
        xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
    <persistence-unit name="vehicle" transaction-type="JTA">
    </persistence-unit>
</persistence>
```

Кода, показанного в этом примере, достаточно для большинства корпоративных приложений.

В следующем фрагменте демонстрируется файл `persistence.xml`, содержащий несколько описаний блоков управления данными для нескольких источников данных:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">

    <persistence-unit name="vehicle" transaction-type="JTA">
        <jta-data-source>jdbc/VehicleDB</jta-data-source>
    </persistence-unit>
    <persistence-unit name="order" transaction-type="JTA">
        <jta-data-source>jdbc/OrderDB</jta-data-source>
    </persistence-unit>
</persistence>
```

При внедрении диспетчеров сущностей нужно указывать желаемый блок управления данными по его имени. Диспетчеры сущностей всегда соответствуют

определенному контексту сохранения данных, который использует один и тот же блок управления данными. В следующем определении `CarManagement` показан предыдущий пример в среде с несколькими блоками управления данными:

```
@Stateless
public class CarManagement {

    @PersistenceContext(unitName = "vehicle")
    EntityManager entityManager;

    public void replaceEngine(long carIdentifier, Engine engine) {
        Car car = entityManager.find(Car.class, carIdentifier);
        car.replaceEngine(engine);

        // операция слияния применяется для объекта "автомобиль"
        // и всех каскадных отношений
        entityManager.merge(car);
    }
}
```

Иногда внедрение диспетчеров объектов можно упростить, воспользовавшись полями CDI-генератора. Если диспетчеры сущностей создаются явно, с применением специальных квалификаторов, то внедрение может быть реализовано в виде следующего файла:

```
public class EntityManagerExposer {

    @Produces
    @VehicleDB
    @PersistenceContext(unitName = "vehicle")
    private EntityManager vehicleEntityManager;

    @Produces
    @OrderDB
    @PersistenceContext(unitName = "order")
    private EntityManager orderEntityManager;
}
```

Теперь можно внедрить диспетчеры созданных сущностей с помощью `@Inject` и квалификатора типов:

```
public class CarManagement {

    @Inject
    @VehicleDB
    EntityManager entityManager;

    …
}
```

Такой подход упрощает использование данных в средах, где в разных местах внедряются разные диспетчеры сущностей.

Есть и другие варианты преобразования моделей предметной области в базы данных. Преобразование базы данных также может быть определено в XML-

файлах. Однако прежние подходы, принятые в J2EE, показали, что декларативная конфигурация с использованием аннотаций более продуктивна. Аннотирование моделей предметной области также обеспечивает лучшее представление о процессе.

Транзакции

Операции по взаимодействию с базой данных необходимо выполнять в виде транзакций. Изменение управляемых сущностей и их синхронизация с источником данных должны реализовываться в рамках транзакции. Таким образом, транзакция охватывает действие модификации и, как правило, весь бизнес-сценарий.

Если контур реализован как EJB, то по умолчанию транзакция является активной во время выполнения бизнес-метода. Это соответствует типичным используемым в приложении сценариям сохранения данных JPA.

Такое же поведение реализуется с помощью управляемых CDI-объектов, методы которых аннотируются посредством `@Transactional`. Транзакционные контуры определяют поведение после ввода бизнес-метода. По умолчанию это поведение `REQUIRED`: транзакция либо создается, либо повторно применяется, если вызывающий контекст уже выполнен в активной транзакции.

При поведении `REQUIRES_NEW` всегда будет запускаться новая транзакция, которая реализуется индивидуально и может возобновить предыдущую транзакцию после того, как завершит работу метод новой транзакции. Это полезно для длительных бизнес-процессов, обслуживающих большое количество данных, которые могут обрабатываться несколькими отдельными транзакциями.

Возможны и другие варианты поведения транзакций, такие как принудительное выполнение уже активной транзакции или полное отсутствие поддержки транзакций. Они настраиваются в виде аннотаций бизнес-методов с помощью `@Transactional`. EJB-объекты неявно определяют транзакции с поведением `REQUIRED`.

Системы реляционных баз данных хорошо интегрируются в приложения Java EE. В соответствии с соглашением о конфигурации в них изначально предусматриваются типичные варианты использования.

Реляционные базы данных или NoSQL?

За последние несколько лет в технологии баз данных произошли большие изменения, особенно в отношении распространения. Однако традиционные реляционные базы данных по-прежнему остаются самым популярным вариантом. Их наиболее важными характеристиками являются табличные структуры данных и поведение с поддержкой транзакций.

В системах баз данных *NoSQL* (*«не SQL»* или *«не только SQL»*) данные представлены в формах, отличных от реляционных таблиц, таких как документные системы управления данными, хранилища типа «ключ — значение», столбцовые хранилища и базы данных графов. Большинство из них жертвуют согласованностью в пользу доступности, масштабируемости и устойчивости к разделению

сети. В основе NoSQL лежит идея отказа от полной поддержки реляционных табличных структур, *ACID*-транзакций (*Atomicity*, *Consistency*, *Isolation*, *Durability* — атомарность, согласованность, изолированность, долговечность) и внешних ключей, а также объединений таблиц. Взамен получают возможность горизонтального масштабирования. Этот принцип восходит к известной теореме *CAP* (*Consistency*, *Availability*, *Partition tolerance* — согласованность, доступность, устойчивость к разделению), которая гласит, что для распределенных хранилищ данных невозможно гарантировать более двух из трех указанных ограничений. Поскольку распределенные сети не являются надежными (устойчивыми к разделению), то можно выбирать, будет система гарантировать согласованность или горизонтальную масштабируемость. Большинство баз данных NoSQL согласованности предпочитают масштабируемость. Необходимо учитывать это при выборе технологии хранения данных.

Причина использования систем NoSQL кроется в недостатках реляционных баз данных. Самая большая проблема реляционных баз данных, поддерживающих ACID, заключается в невозможности горизонтального масштабирования. К базам данных, лежащим в основе корпоративных систем, обычно обращаются несколько серверов приложений. Данные, которые необходимо постоянно обновлять, должны синхронизироваться с центральным хранилищем. Синхронизация выполняется как техническая транзакция бизнес-сценария. При репликации базы данных должны сохранять согласованность и поддерживать между собой распределенные транзакции. Однако распределенные транзакции не масштабируются и не всегда надежно работают в различных решениях.

Тем не менее реляционные базы данных довольно хорошо подходят для большинства корпоративных приложений. Если же требуется горизонтальное масштабирование и централизованная база данных больше не подходит, можно разделить хранилище, воспользовавшись такой технологией, как событийно-ориентированная архитектура. Мы рассмотрим эту тему подробнее в главе 8.

У баз данных NoSQL тоже есть недостатки, особенно в отношении поведения с поддержкой транзакционной семантики. Должны ли данные храниться в системе с поддержкой транзакций, зависит от бизнес-требований приложения. Как показывает опыт, почти во всех корпоративных системах важно гарантировать сохраняемость данных— иными словами, использовать транзакции. Однако иногда встречаются особые категории данных. В то время как одни модели предметной области являются более важными и требуют обработки транзакций, в других данные могут быть вычислены повторно или восстановлены. К таким данным относятся статистика, рекомендации или информация, сохраненная в кэше. Для последнего типа данных хранилища NoSQL могут быть хорошим вариантом.

На момент написания этой книги ни одна система NoSQL не стала фактическим стандартом. Многие из них сильно различаются по концепциям и практике применения. Не существует также стандартной NoSQL, включенной в комплект Java EE 8. Поэтому доступ к системам NoSQL обычно реализуется с использованием API Java от сторонних поставщиков. Они задействуют стандарты более низкого уровня, такие как JDBC, или собственные API.

Сквозные задачи

В корпоративных приложениях приходится решать ряд технически обоснованных сквозных задач. Примерами являются транзакции, протоколирование, кэширование, обеспечение устойчивости, мониторинг, обеспечение безопасности и другие нефункциональные требования. Даже в системах, предназначенных исключительно для бизнеса, сценарии использования требуют небольшой доработки.

Пример нефункциональной сквозной задачи мы только что рассмотрели — это обработка транзакций. В Java EE инженерам не приходится тратить много времени и усилий на интеграцию поведения с поддержкой транзакционной семантики. Это верно и в отношении других сквозных задач.

Ярким примером сквозных задач являются перехватчики Java EE. В соответствии с концепцией аспектно-ориентированного программирования реализация сквозной задачи должна быть отделена от декорированной функциональности. Методы управляемых объектов могут быть оформлены путем определения перехватчиков, прерывающих работу приложения и выполняющих требуемую задачу. Перехватчики полностью контролируют реализацию перехваченного метода, включая возвращаемые значения и генерируемые исключения. Для того чтобы соответствовать стилю других API, перехватчики интегрированы просто, без особых ограничений на декорированный функционал.

Применение перехватчика продемонстрировано в рассмотренном ранее примере прозрачной обработки ошибок в клиентском HTTP-классе. Бизнес-методы также могут быть декорированы с использованием нестандартных привязок перехватчиков. В следующем примере показан обоснованный бизнес-сценарием процесс отслеживания, реализованный посредством специальных аннотаций:

```
@Stateless
public class CarManufacturer {

    …

    @Tracked(ProcessTracker.Category.MANUFACTURER)
    public Car manufactureCar(Specification spec) {
        …
    }
}
```

Аннотация `Tracked` определяет так называемую привязку перехватчика. Параметр аннотации представляет собой необязательное значение для настройки параметров перехватчика:

```
import javax.enterprise.util.Nonbinding;
import javax.interceptor.InterceptorBinding;

@InterceptorBinding
@Inherited
@Documented
@Target({TYPE, METHOD})
@Retention(RUNTIME)
```

```
public @interface Tracked {

    @Nonbinding
    ProcessTracker.Category value();
}
```

Перехватчик активизируется с помощью аннотации привязки:

```
import javax.annotation.Priority;

@Tracked(ProcessTracker.Category.UNUSED)
@Interceptor
@Priority(Interceptor.Priority.APPLICATION)
public class TrackingInterceptor {

    @Inject
    ProcessTracker processTracker;

    @AroundInvoke
    public Object aroundInvoke(InvocationContext context) throws Exception {
        Tracked tracked = resolveAnnotation(context);

        if (tracked != null) {
            ProcessTracker.Category category = tracked.value();
            processTracker.track(category);
        }

        return context.proceed();
    }

    private Tracked resolveAnnotation(InvocationContext context) {
        Function<AnnotatedElement, Tracked> extractor =
            c->c.getAnnotation(Tracked.class);
        Method method = context.getMethod();

        Tracked tracked = extractor.apply(method);
        return tracked != null ? tracked :
            extractor.apply(method.getDeclaringClass());
    }
}
```

По умолчанию перехватчики, объединенные посредством привязок, не активизированы. Перехватчик должен либо быть явно активизирован путем указания приоритета в аннотации `@Priority`, как показано в этом примере, либо активизирован в дескрипторе `beans.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
        bean-discovery-mode="all">
    <interceptors>
        <class>com.example.cars.processes.TrackingInterceptor</class>
    </interceptors>
</beans>
```

Для извлечения возможных параметров аннотации, таких как категория отслеживания процессов в данном примере, перехватчики могут использовать отражение. Привязки перехватчиков могут размещаться либо на уровне метода, либо на уровне класса.

Перехватчики декорируют поведение методов, не образуя тесной связи с ними. Они особенно полезны для таких сценариев, где существуют сквозные задачи с большим количеством функциональных возможностей.

Перехватчики аналогичны CDI-декораторам. Обе концепции декорируют управляемые объекты с нестандартным поведением, которое инкапсулируется в другом месте. Разница заключается в том, что декораторы предназначены для бизнес-логики, которая в основном характерна для декорированного управляемого объекта. В отличие от них перехватчики используются главным образом для решения технических задач. Они допускают более широкое применение, позволяя аннотировать все виды управляемых объектов. Обе концепции являются полезной функциональностью для решения сквозных задач.

Настройка приложений

Если приложение не может быть жестко запрограммировано, а должно определяться динамически, его поведение можно настроить. Реализация настройки зависит от конкретного приложения и характера его динамического поведения.

Какие аспекты приложения необходимо настраивать? Достаточно ли создать файлы конфигурации, которые и так являются частью поставляемого артефакта? Нужно ли настраивать упакованное приложение извне? Есть ли необходимость изменять поведение приложения во время его выполнения?

Настройка, которая не нуждается в изменении после сборки приложения, может быть легко реализована в самом проекте, то есть в исходном коде. Поэтому будем считать, что нам нужна большая гибкость.

Пожалуй, самым простым способом настройки в среде Java является предоставление файлов свойств, содержащих параметры настройки в виде пар «ключ — значение». Оттуда параметры настройки затем извлекаются для применения в коде. Конечно, можно написать компоненты Java, которые бы программно обеспечивали значения свойств. В среде Java EE для извлечения таких компонентов внедряются зависимости.

На момент написания этой книги ни один стандарт Java EE не поддерживал готовую конфигурацию. Однако с помощью CDI-функций можно реализовать эту функциональность буквально в нескольких строках кода. Далее показано возможное решение, позволяющее внедрять значения конфигурации, идентифицируемые по ключам:

```
@Stateless
public class CarManufacturer {

    @Inject
    @Config("car.default.color")
```

```
    String defaultColor;

    public Car manufactureCar(Specification spec) {
        // использовать defaultColor
    }
}
```

Для того чтобы однозначно внедрять параметры конфигурации, например предоставляемые в виде строк, нужно задействовать квалификатор, такой как `@Config`. Этот специальный квалификатор определен и в нашем приложении. Его назначение состоит в том, чтобы внедрять значения, идентифицированные предоставленным ключом:

```
@Qualifier
@Documented
@Retention(RUNTIME)
public @interface Config {

    @Nonbinding
    String value();
}
```

CDI-генератор отвечает за получение и предоставление определенных параметров конфигурации:

```
import javax.enterprise.inject.spi.InjectionPoint;
import java.io.*;
import java.util.Properties;

@ApplicationScoped
public class ConfigurationExposer {

    private final Properties properties = new Properties();

    @PostConstruct
    private void initProperties() {
        try (InputStream inputStream = ConfigurationExposer.class
                .getResourceAsStream("/application.properties")) {
            properties.load(inputStream);
        } catch (IOException e) {
            throw new IllegalStateException("Не удалось запустить конфигурацию", e);
        }
    }

    @Produces
    @Config("")
    public String exposeConfig(InjectionPoint injectionPoint) {
        Config config =
            injectionPoint.getAnnotated().getAnnotation(Config.class);
        if (config != null)
            return properties.getProperty(config.value());
        return null;
    }
}
```

Ссылочный ключ в аннотации `@Config` является атрибутом без привязки, поскольку все внедренные значения обрабатываются CDI-генератором. Объект `InjectionPoint`, предоставленный CDI, содержит информацию о месте, в котором описано внедрение зависимости. Генератор получает аннотацию с фактическим ключом конфигурации и использует ее для поиска соответствующего параметра настройки. Предполагается, что файл свойств `application.properties` будет находиться там же, где и классы. Таким образом, создаются параметры настройки, которые должны быть доступны во время выполнения приложения. Поскольку карта свойств инициируется только один раз, после загрузки значения изменяться не будут. Управляемый объект, читающий параметры настройки, является видимым в пределах всего приложения и доступен только для однократной загрузки необходимых значений в карту свойств.

Если сценарий требует изменения параметров настройки во время реализации приложения, то метод-генератор должен перезагрузить файл конфигурации. Область видимости метода-генератора определяет жизненный цикл параметров настройки и то, как часто будет вызываться этот метод.

В данном примере реализована настройка с применением простой Java EE. Существует ряд сторонних CDI-расширений, предоставляющих аналогичные, а также более сложные функции. На момент написания этой книги популярным примером такого решения был Apache Deltaspike.

Помимо корпоративной технологии, следует принять во внимание еще один важный фактор — информационную среду, в которой работает контейнер, в первую очередь потому, что технологии контейнеров накладывают определенные ограничения на среду выполнения приложения. Более подробно современные информационные среды и их влияние на реализацию приложений Java EE, включая проектирование динамической конфигурации, будут рассмотрены в главе 5.

Главное преимущество CDI-генераторов заключается в их гибкости. Они позволяют легко подключить любой источник параметров настройки и назначить нужные параметры конфигурации.

Кэширование

Кэширование — это технически обоснованная сквозная задача, которая приобретает большое значение, когда приложения сталкиваются с проблемами производительности (медленными внешними системами, сложными кэшируемыми вычислениями или огромными объемами данных). В общем случае кэширование нацелено на снижение времени отклика за счет сохранения данных, которые сложно получать повторно, в потенциально более быстром кэше. Типичным примером использования кэша является сохранение в памяти ответов, полученных из внешних систем или из баз данных.

Прежде чем реализовать кэширование, следует задать такие вопросы: необходимо ли оно и возможно ли в принципе? Некоторые данные, например вычисляемые по требованию, не могут быть кэшированы. Но даже если ситуация

и данные допускают кэширование, следует подумать, нет ли иного решения. Кэширование подразумевает дублирование, и есть вероятность получить устаревшую информацию. Вообще говоря, в большинстве корпоративных приложений кэширования следует избегать. Например, если база данных работает слишком медленно, рекомендуется изучить другие варианты решения этой проблемы, такие как индексирование.

Какие именно решения кэширования требуются, зависит от ситуации. В целом одно лишь кэширование непосредственно в памяти приложения решает много проблем.

Самый простой способ кэширования информации — хранить ее в одном месте в приложении. Для этого идеально подходят управляемые объекты в виде синглтонов. Структура данных, которая естественным образом соответствует назначению кэша, — это тип Java `Map`.

Представленный ранее пример с фрагментом кода `CarStorage` — это EJB-синглтон с параллелизмом, управляемым этим объектом. Данный синглтон содержит ориентированную на многопоточность карту хранения данных. Это хранилище внедряется в другие управляемые объекты и используется там:

```
@Singleton
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class CarStorage {

    private final Map<String, Car> cars = new ConcurrentHashMap<>();

    public void store(Car car) {
        cars.put(car.getId(), car);
    }

    public Car retrieve(String id) {
        return cars.get(id);
    }
}
```

Если требуется больше гибкости, например нужна предварительная загрузка содержимого кэша из файла, то управляемый объект может контролировать жизненный цикл, используя методы, реализуемые сразу после создания и непосредственно перед уничтожением объекта. Для того чтобы гарантировать выполнение функций во время запуска приложения, EJB снабжается аннотацией `@Startup`:

```
@Singleton
@Startup
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class CarStorage {
    …

    @PostConstruct
    private void loadStorage() {
        // загрузка содержимого файла
    }

    @PreDestroy
```

```
    private void writeStorage() {
        // запись данных в файл
    }
}
```

Перехватчик может использоваться для прозрачного добавления кэша без необходимости его программного внедрения и применения. Перехватчик прерывает выполнение приложения до вызова бизнес-метода и вместо этого возвращает кэшированные значения. Наиболее ярким примером является функциональность `CacheResult` из *Java Temporary Caching API (JCache)*. JCache — стандарт, предназначенный для Java EE, но на момент написания этой книги еще не включенный в обобщающую спецификацию. В приложениях, в которых используется функциональность JCache, подходящие для этого бизнес-методы снабжаются аннотацией `@CacheResult` и прозрачно обслуживаются соответствующим кэшем.

В целом JCache позволяет выполнять сложное кэширование для тех сценариев, где простых решений Java EE недостаточно. В частности, реализации JCache обеспечивают возможность распределенного кэширования. В настоящее время обычно используются такие решения, как *Hazelcast*, *Infinispan* и *Ehcache*. Это особенно важно в тех случаях, когда необходимо интегрировать несколько кэшей, каждый из которых предназначен для решения своих задач, таких как замещение кэша. Технология JCache и ее реализации предоставляют подобные сильные решения.

Последовательность выполнения

Бизнес-процессы, реализуемые в корпоративных приложениях, описывают определенные потоки процессов. Для задействованных бизнес-сценариев это либо процесс синхронных запросов и ответов, либо асинхронная обработка инициированного процесса.

Бизнес-сценарии вызываются в отдельных потоках, по одному потоку на запрос или вызов. Потоки создаются контейнером и помещаются в накопитель для повторного использования после того, как вызов был успешно обработан. По умолчанию бизнес-процессы, определенные в классах приложений, а также сквозные задачи, такие как транзакции, осуществляются последовательно.

Синхронное выполнение

Типичный сценарий, когда на HTTP-запрос требуется ответ от базы данных, реализуется следующим образом. Один поток обрабатывает запрос, поступающий в контур, например JAX-RS `UsersResource`, путем инверсии принципа управления; ресурсный метод JAX-RS вызывается контейнером. Ресурс внедряет и использует EJB-объект `UserManagement`, который также неявно вызывается контейнером. Все операции выполняются посредниками синхронно. Для хранения новой сущности `User` EJB будет применять диспетчер сущностей, и как только бизнес-метод,

инициировавший текущую активную транзакцию, закончит работу, контейнер попытается зафиксировать транзакцию в базе данных. В зависимости от результата транзакции ресурсный метод контура возобновляет работу и формирует ответ клиенту. Все происходит синхронно, в это время клиент заблокирован и ожидает ответа.

Синхронное выполнение включает в себя обработку синхронных CDI-событий. Они отделяют запуск событий предметной области от их обработки, однако обрабатываются события синхронно. Существует несколько методов наблюдения за транзакциями. Если указан этап транзакции, то событие может быть обработано на этом этапе — во время фиксации транзакции, до ее завершения, после завершения, в случае неудачной либо успешной транзакции. По умолчанию или если транзакция неактивна CDI-события обрабатываются сразу при их возникновении. Это позволяет инженерам реализовывать сложные решения — например, с использованием событий, которые происходят только после успешного добавления сущностей в базу данных. Как бы то ни было, во всех случаях обработка выполняется синхронно.

Асинхронное выполнение

Синхронное выполнение задач удовлетворяет требованиям многих бизнес-сценариев, но бывают случаи, когда нужно асинхронное поведение. В среде Java EE установлен ряд ограничений на использование приложением потоков. Контейнер управляет ресурсами и потоками и помещает их в накопитель. Внешние утилиты контроля параллелизма находятся вне контейнера, и им ничего не известно об этих потоках. Поэтому код приложения не должен запускаться и управлять своими потоками. Для этого он задействует функции Java EE. Существует несколько API со встроенной поддержкой асинхронности.

Асинхронные EJB-методы

Самый простой способ реализации асинхронного поведения — использовать аннотацию `@Asynchronous` для бизнес-метода EJB или EJB-класса. Вызовы этих методов сразу возвращаются, иногда с ответом типа `Future`. Они выполняются в отдельном потоке, управляемом контейнером. Такой способ хорошо работает для простых сценариев, но ограничен EJB-объектами:

```
import javax.ejb.Asynchronous;

@Asynchronous
@Stateless
public class Calculator {

    public void calculatePi(long decimalPlaces) {
        // этот метод может долго выполняться
    }
}
```

Сервис управления выполнением

Для асинхронного выполнения задач в управляемых CDI-объектах или с помощью утилит контроля параллелизма Java SE в состав Java EE включены управляемые контейнером версии функций `ExecutorService` и `ScheduledExecutorService`. Они используются для реализации асинхронных задач в потоках, управляемых контейнерами. Экземпляры `ManagedExecutorService` и `ManagedScheduledExecutorService` внедряются в код приложения. Они могут применяться для выполнения собственной логики, однако наиболее эффективны при объединении с утилитами контроля параллелизма Java SE, такими как дополняемые будущие значения. В следующем примере показано создание дополняемых будущих значений с использованием потоков, управляемых контейнером:

```
import javax.annotation.Resource;
import javax.enterprise.concurrent.ManagedExecutorService;
import java.util.Random;
import java.util.concurrent.CompletableFuture;

@Stateless
public class Calculator {

    @Resource
    ManagedExecutorService mes;

    public CompletableFuture<Double> calculateRandomPi(int
        maxDecimalPlaces) {
        return CompletableFuture.supplyAsync(() -> new
            Random().nextInt(maxDecimalPlaces) + 1, mes)
                .thenApply(this::calculatePi);
    }

    private double calculatePi(long decimalPlaces) {
        …
    }
}
```

Объект `Calculator` возвращает *дополняемое будущее значение типа* `double`, которое еще может быть вычислено при возобновлении работы вызывающего контекста. Его можно запросить, когда вычисления будут закончены, а также объединить с последующими вычислениями. Независимо от того, где в корпоративном приложении требуются новые потоки, для управления ими следует использовать функциональность Java EE.

Асинхронные CDI-события

CDI-события также могут обрабатываться асинхронно. В этом случае контейнер тоже предоставляет поток для обработки событий. Для описания асинхронного обработчика событий метод снабжается аннотацией `@ObservesAsync`, а событие

активизируется с помощью метода `fireAsync()`. В следующих фрагментах кода продемонстрированы асинхронные события CDI:

```
@Stateless
public class CarManufacturer {

    @Inject
    CarFactory carFactory;

    @Inject
    Event<CarCreated> carCreated;

    public Car manufactureCar(Specification spec) {
        Car car = carFactory.createCar(spec);
        carCreated.fireAsync(new CarCreated(spec));
        return car;
    }
}
```

Обработчик события вызывается в собственном потоке, управляемом контейнером:

```
import javax.enterprise.event.ObservesAsync;

public class CreatedCarListener {

    public void onCarCreated(@ObservesAsync CarCreated event) {
        // асинхронная обработка события
    }
}
```

Из соображений обратной совместимости синхронные CDI-события также могут обрабатываться в асинхронном EJB-методе. Таким образом, события и обработчики определяются как синхронные, а метод обработчика является бизнес-методом EJB с аннотацией `@Asynchronous`. До того как асинхронные события внесли в стандарт CDI для Java EE 8, это был единственный способ реализовать данную функцию. Во избежание путаницы в Java EE 8 и следующих версиях такой реализации лучше избегать.

Области видимости при асинхронной обработке

Поскольку контейнер не имеет информации о том, как долго могут выполняться асинхронные задачи, использование областей видимости в этом случае ограничено. Объекты с областью видимости в пределах запроса или сессии, которые были доступны при запуске асинхронной задачи, не обязательно будут активными в течение всей ее реализации — запрос и сессия могут закончиться задолго до ее завершения. Таким образом, потоки, выполняющие асинхронные задачи, например предоставляемые службой запланированных исполнителей или асинхронными событиями, могут не иметь доступа к экземплярам управляемых объектов с областью видимости в пределах запроса или сессии, которые были активны во время вызова. То же самое касается доступа к ссылкам на внедренные экземпляры, например в лямбда-методах, которые являются частью синхронного выполнения.

Это необходимо учитывать при моделировании асинхронных задач. Вся информация о конкретном вызове должна быть предоставлена во время запуска задачи. Однако асинхронная задача может иметь собственные экземпляры управляемых объектов с ограниченной областью видимости.

Выполнение в заданное время

Бизнес-сценарии могут вызываться не только извне, например, по HTTP-запросу, но и изнутри приложения — задачей, запускаемой в определенное время.

В мире Unix популярна функциональность для запуска периодических заданий — это задачи планировщика. EJB-объекты обеспечивают аналогичные возможности с использованием EJB-таймеров. Таймеры вызывают бизнес-методы через заданные интервалы или по истечении определенного времени. В следующем примере представлено описание циклического таймера, который запускается каждые десять минут:

```
import javax.ejb.Schedule;
import javax.ejb.Startup;

@Singleton
@Startup
public class PeriodicJob {

    @Schedule(minute = "*/10", hour = "*", persistent = false)
    public void executeJob() {
        // выполняется каждые 10 минут
    }
}
```

Любые EJB-объекты — синглтоны, управляемые объекты с сохранением или без сохранения состояния — могут создавать таймеры. Однако в большинстве сценариев имеет смысл создавать таймеры только для синглтонов. Задержка устанавливается для всех активных объектов. Обычно она нужна, чтобы вовремя запускать запланированные задачи, именно поэтому она используется в синглтоне. По этой же причине в данном примере EJB-объект должен быть активным при запуске приложения. Это гарантирует, что таймер сразу начнет работать.

Если описать таймер как постоянный, то его время жизни распространится на весь жизненный цикл JVM. Контейнер отвечает за сохранение постоянных таймеров, обычно в базе данных. Постоянные таймеры, которые должны работать в то время, пока приложение недоступно, включаются при запуске. Это также позволяет использовать одни и те же таймеры несколькими экземплярами объекта. Постоянные таймеры при соответствующей конфигурации сервера — подходящее решение, если нужно выполнить бизнес-процесс ровно один раз на нескольких серверах.

Таймеры, которые создаются автоматически с помощью аннотации `@Schedule`, описываются с помощью Unix-подобных cron-выражений. Для большей гибкости EJB-таймеры описываются программно с помощью предоставляемой контейнером службы таймера, которая создает методы обратного вызова `Timers` и `@Timeout`.

Периодические и отложенные задачи также могут быть описаны за пределами EJB-компонентов с помощью управляемой контейнером службы запланированных

исполнителей. Экземпляр `ManagedScheduledExecutorService`, выполняющий задачи по истечении указанной задержки или с заданной периодичностью, внедряется в управляемые компоненты. Эти задачи будут реализовываться в потоках, управляемых контейнерами:

```
@ApplicationScoped
public class Periodic {

    @Resource
    ManagedScheduledExecutorService mses;

    public void startAsyncJobs() {
        mses.schedule(this::execute, 10, TimeUnit.SECONDS);
        mses.scheduleAtFixedRate(this::execute, 60, 10, TimeUnit.SECONDS);
    }

    private void execute() {
        …
    }
}
```

Вызов метода `startAsyncJobs()` приведет к запуску функции `execute()` в управляемом потоке через десять секунд после вызова и затем каждые десять секунд по прошествии первой минуты.

Асинхронность и реактивность в JAX-RS

JAX-RS поддерживает асинхронное поведение, чтобы излишне не блокировать потоки запросов на стороне сервера. Даже если HTTP-соединение ожидает ответа, поток запросов может продолжать обрабатывать другие запросы, пока на сервере протекает длительный процесс. Потоки запросов объединяются в контейнере, и это хранилище запросов имеет определенный размер. Чтобы не занимать понапрасну поток запросов, асинхронные ресурсные методы JAX-RS создают задачи, которые выполняются при возврате потока запроса и могут быть использованы повторно. HTTP-соединение возобновляется и выдает отклик после завершения асинхронной задачи или по истечении времени ожидания. В следующем примере показан асинхронный ресурсный метод JAX-RS:

```
@Path("users")
@Consumes(MediaType.APPLICATION_JSON)
public class UsersResource {

    @Resource
    ManagedExecutorService mes;

    …

    @POST
    public CompletionStage<Response> createUserAsync(User user) {
        return CompletableFuture.supplyAsync(() -> createUser(user), mes);
    }

    private Response createUser(User user) {
```

```
        userStore.create(user);

        return Response.accepted().build();
    }
}
```

Для того чтобы поток запросов не был занят слишком долго, метод JAX-RS должен быстро завершаться. Это связано с тем, что ресурсный метод вызывается из контейнера посредством инверсии управления. Результат, полученный на этапе завершения, будет использован для возобновления клиентского соединения по окончании обработки.

Возврат этапов завершения — сравнительно новая технология в API JAX-RS. Если нужно описать задержку и при этом обеспечить большую гибкость при асинхронном ответе, то в метод можно включить тип `AsyncResponse`. Этот подход продемонстрирован в следующем примере:

```
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;

@Path("users")
@Consumes(MediaType.APPLICATION_JSON)
public class UsersResource {

    @Resource
    ManagedExecutorService mes;

    …

    @POST
    public void createUserAsync(User user, @Suspended AsyncResponse
        response) {

        response.setTimeout(5, TimeUnit.SECONDS);
        response.setTimeoutHandler(r ->
          r.resume(Response.status(Response.Status.SERVICE_UNAVAILABLE).build()));

        mes.execute(() -> response.resume(createUser(user)));
    }
}
```

Благодаря создаваемым тайм-аутам клиентский запрос станет ждать не бесконечно, а только до тех пор, пока не будет получен результат или не истечет время ожидания вызова. Однако вычисления будут продолжаться, поскольку они выполняются асинхронно.

Для ресурсов JAX-RS, реализуемых как EJB-объекты, можно применять аннотацию `@Asynchronous`, чтобы не вызывать асинхронные бизнес-методы явно, через сервис-исполнитель.

Клиент JAX-RS также поддерживает асинхронное поведение. В зависимости от требований имеет смысл не блокировать его при HTTP-вызовах. В предыдущем примере показано, как устанавливать задержки для клиентских запросов. Для длительно выполняемых и особенно параллельных внешних системных вызовов лучше использовать асинхронное и реактивное поведение.

Рассмотрим несколько серверных приложений, предоставляющих информацию о погоде. Клиентский компонент обращается ко всем этим приложениям и вычисляет усредненный прогноз погоды. В идеале можно было бы сделать доступ к системам параллельным:

```
import java.util.stream.Collectors;

@ApplicationScoped
public class WeatherForecast {

    private Client client;
    private List<WebTarget> targets;

    @Resource
    ManagedExecutorService mes;

    @PostConstruct
    private void initClient() {
        client = ClientBuilder.newClient();
        targets = …
    }

    public Forecast getAverageForecast() {
        return invokeTargetsAsync()
                .stream()
                .map(CompletableFuture::join)
                .reduce(this::calculateAverage)
                .orElseThrow(() -> new IllegalStateException("Нет доступных
                    прогнозов погоды"));
    }

    private List<CompletableFuture<Forecast>> invokeTargetsAsync() {
        return targets.stream()
                .map(t -> CompletableFuture.supplyAsync(() -> t
                        .request(MediaType.APPLICATION_JSON_TYPE)
                        .get(Forecast.class), mes))
                .collect(Collectors.toList());
    }

    private Forecast calculateAverage(Forecast first, Forecast second) {
        …
    }

    @PreDestroy
    public void closeClient() {
        client.close();
    }
}
```

Метод `invokeTargetsAsync()` вызывает доступные объекты асинхронно, задействуя службу запланированных исполнителей. Дескрипторы `CompletableFuture` возвращаются и используются для вычисления усредненных результатов. Запуск метода `join()` будет блокироваться до тех пор, пока вызов не завершится и не будут получены результаты.

Объекты, вызываемые асинхронно, запускаются и ожидают отклика сразу от нескольких ресурсов, возможно, более медленных. В этом случае ожидание ответов от ресурсов метеослужбы занимает столько времени, сколько приходится ожидать самого медленного отклика, а не всех откликов вместе.

В последнюю версию JAX-RS встроена поддержка этапов завершения, что позволяет сократить стереотипный код в приложениях. Как и в случае с дополняемыми значениями, вызов сразу возвращает код этапа завершения для дальнейшего использования. В следующем примере показаны реактивные клиентские функции JAX-RS с применением вызова `rx()`:

```
public Forecast getAverageForecast() {
    return invokeTargetsAsync()
            .stream()
            .reduce((l, r) -> l.thenCombine(r, this::calculateAverage))
            .map(s -> s.toCompletableFuture().join())
            .orElseThrow(() -> new IllegalStateException("Нет доступных
                прогнозов погоды"));
}

private List<CompletionStage<Forecast>> invokeTargetsAsync() {
    return targets.stream()
        .map(t -> t
            .request(MediaType.APPLICATION_JSON_TYPE)
                .rx()
                .get(Forecast.class))
            .collect(Collectors.toList());
}
```

В приведенном примере не требуется искать службу запланированных исполнителей — клиент JAX-RS будет управлять этим сам.

До того как появился метод `rx()`, в клиентах применялся явный вызов `async()`. Этот метод вел себя аналогично, но возвращал только объекты `Future`. Использование в клиентах реактивного подхода оптимально для большинства проектов.

Как видите, в среде Java EE задействуется служба исполнителей, управляемая контейнером.

Концепции и принципы проектирования в современной Java EE

API Java EE основан на соглашениях и принципах проектирования, которые прописаны в виде стандартов. Инженеры-программисты встретят в нем знакомые шаблоны API и подходы к разработке приложений. Целью Java EE является поощрение последовательного использования API.

Главный принцип приложений, ориентированных в первую очередь на реализацию бизнес-сценариев, звучит так: *технология не должна мешать*. Как уже упоминалось, у инженеров должна быть возможность сосредоточиться на реализации бизнес-логики, не тратя большую часть времени на технологические

и инфраструктурные вопросы. В идеале логика предметной области реализуется на простом Java и *дополняется* аннотациями и другими свойствами, поддерживаемыми корпоративной средой, не влияя на код предметной области и не усложняя его. Это означает, что технология не требует большого внимания инженеров и не накладывает слишком больших ограничений. Раньше среда J2EE требовала множества очень сложных решений. Для реализации интерфейсов и расширения базовых классов приходилось использовать управляемые объекты и объекты постоянного хранения. Это усложняло логику предметной области и затрудняло тестирование.

В Java EE логика предметной области реализуется в виде простых классов Java, снабженных аннотациями, согласно которым контейнер в процессе выполнения приложения решает те или иные корпоративные задачи. Практика создания чистого кода часто предполагает написание кода скорее красивого, нежели удобного для повторного использования. Java EE поддерживает этот подход. Если по какой-то причине нужно убрать технологию и оставить чистую логику предметной области, это делается простым удалением соответствующих аннотаций.

Как мы увидим в главе 7, такой подход к программированию подразумевает необходимость тестирования, поскольку для программистов большинство спецификаций Java EE — это не более чем аннотации.

Во всем API принят принцип проектирования, называемый *инверсией управления (inversion of control, IoC)*, — другими словами, «не звоните нам, мы сами позвоним». Это особенно заметно в контурах приложений, таких как ресурсы JAX-RS. Ресурсные методы описываются с помощью аннотаций Java-методов, которые позднее вызываются контейнером в соответствующем контексте. То же самое справедливо и для внедрения зависимостей, при которых приходится выбирать генераторы или учитывать такие сквозные задачи, как перехватчики. Разработчики приложений могут сосредоточиться на воплощении в жизнь логики и описании отношений, оставив реализацию технических деталей в контейнере. Еще один пример, не столь очевидный, — это описание преобразования Java-объектов в JSON и обратно посредством аннотаций JSON-B. Объекты преобразуются не только в явном, запрограммированном виде, но и неявно, в декларативном стиле.

Еще один принцип, который позволяет инженерам эффективно применять эту технологию, — *программирование по соглашениям*. По умолчанию в Java EE задано определенное поведение, соответствующее большинству сценариев использования. Если его недостаточно или оно не соответствует требованиям, поведение можно переопределить, часто на нескольких уровнях.

Есть множество примеров программирования по соглашению. Один из них — применение ресурсных методов JAX-RS, преобразующих функционал Java в HTTP-отклики. Если стандартное поведение JAX-RS по отношению к откликам не удовлетворяет требованиям, можно применять тип возвращаемого ответа `Response`. Другим примером является спецификация управляемых объектов, которая обычно реализуется с помощью аннотаций. Для того чтобы изменить это поведение, можно задействовать XML-дескриптор `beans.xml`. Для программистов очень удобно, что в современном мире Java EE корпоративные приложения разрабатываются прагматичным и высокопроизводительным способом, который обычно не требует такого интенсивного использования XML, как прежде.

Что же касается продуктивности работы программистов, то еще один важный принцип разработки на Java EE состоит в том, что эта платформа требует интеграции в контейнере различных стандартов. Поскольку контейнеры поддерживают определенный набор API — а в случае поддержки всего API Java EE это именно так, — требуется также, чтобы реализации API обеспечивали простую интеграцию других API. Достоинство этого подхода — возможность использования ресурсами JAX-RS преобразования JSON-B и технологии Bean Validation без дополнительной явной настройки, за исключением аннотаций. В предыдущих примерах мы увидели, как функции, определенные в отдельных стандартах, можно применять совместно, не прилагая дополнительных усилий. Это одно из самых больших преимуществ платформы Java EE. Обобщающая спецификация гарантирует сочетание отдельных стандартов между собой. Программисты могут полагаться на определенные функции и реализацию, предоставляемые сервером приложений.

Удобный в сопровождении высококачественный код

Программисты в целом согласны с тем, что следует стремиться писать код высокого качества. Однако не все технологии одинаково хорошо подходят для этого.

Как уже упоминалось в начале книги, в центре внимания при разработке приложений должна быть бизнес-логика. В случае изменений бизнес-логики или появления новых знаний необходимо обновить модель предметной области, а также исходный код. Для создания и поддержки высококачественной модели предметной области и исходного кода в целом требуется итеративный рефакторинг. Усилия, направленные на углубление понимания предметной области, описаны в концепции проблемно-ориентированного проектирования.

Есть много литературы, посвященной рефакторингу на уровне кода. После того как бизнес-логика будет представлена в виде кода и проверена тестами, программистам следует потратить некоторое время и приложить усилия к тому, чтобы переосмыслить и улучшить первый вариант. Это касается идентификаторов имен, методов и классов. Особенно важны *выбор имен*, *уровни абстракций* и *единые точки ответственности*.

Согласно определению проблемно-ориентированного проектирования предметная область должна максимально соответствовать своему представлению в виде кода. Сюда входит, в частности, язык предметной области — другими словами, то, как программисты и бизнес-эксперты говорят об определенных функциях. Цель всей команды — найти *универсальный общий язык*, который будет эффективно использоваться не только в дискуссиях и на презентационных слайдах, но и в коде. Уточнение знаний в области бизнеса будет происходить циклически. Как и рефакторинг на уровне кода, этот подход подразумевает, что первоначальная модель не будет полностью соответствовать всем требованиям.

Таким образом, применяемая технология должна поддерживать изменения модели и кода. Если ограничений слишком много, то вносить изменения впоследствии будет трудно.

Для разработки приложений в целом, и особенно для рефакторинга, крайне важно, чтобы программное обеспечение было достаточно охвачено автоматизированными тестами. Поскольку код постоянно изменяется, регрессионные тесты гарантируют, что ни одна из бизнес-функций не будет при этом случайно повреждена. Таким образом, достаточное количество контрольных тестов поддерживает рефакторинг, позволяя инженерам ясно понять, что после внесения изменений весь функционал по-прежнему работает так, как предполагается. В идеале технология должна поддерживать возможность тестирования, не накладывая ограничений на структуру кода. Подробнее мы обсудим это в главе 7.

Для обеспечения возможности рефакторинга слабое связывание предпочтительнее, чем тесное. Изменение одного компонента затрагивает все функции, которые явно его вызывают, и все компоненты, в которых он нуждается. Java EE поддерживает несколько вариантов слабого связывания: внедрение зависимостей, события и сквозные задачи, такие как перехватчики. Все это упрощает изменение кода.

Существует ряд инструментов и методов для измерения качества. В частности, статический анализ кода позволяет собирать информацию о сложности, связности, зависимостях между классами и пакетами и реализации в целом. Эти средства помогают инженерам выявлять потенциальные проблемы и формировать общую картину программного проекта. В главе 6 будет показано, как автоматически проверить качество кода.

В целом рекомендуется постоянно реорганизовывать код и улучшать его качество. Программные проекты часто создаются для внедрения новых функций, приносящих доход, а не для того, чтобы улучшить существующий функционал. Проблема в том, что рефакторинг и повышение качества кода на первый взгляд не приносят пользы для бизнеса. Это, конечно же, не так. Для того чтобы добиться стабильной скорости и интегрировать новые функции с удовлетворительным качеством, необходимо пересмотреть существующие функции. В идеале следует встроить циклы рефакторинга в схему проекта. Как показывает опыт, руководители проектов часто не знают об этой проблеме. Однако команда инженеров-программистов несет ответственность за поддержание качества.

Резюме

Инженерам рекомендуется в первую очередь сосредоточиться на предметной области и бизнес-логике, начиная с контуров бизнес-сценариев, а затем идти вглубь по уровням абстракции. Для реализации простой бизнес-логики применяются такие компоненты ядра Java EE, как EJB, CDI, CDI-генераторы и события. Другие API Java EE в основном задействуются для поддержки технических потребностей бизнес-логики. Как мы увидели, в Java EE на современном уровне реализованы и поощряются многочисленные шаблоны проектирования ПО, а также принципы проблемно-ориентированного проектирования.

Мы научились выбирать и реализовывать синхронную и асинхронную связь и узнали, что выбор технологии связи зависит от бизнес-требований. Особенно

широко используется и хорошо поддерживается в Java EE технология HTTP — через JAX-RS. Ярким примером архитектурного стиля протокола связи, поддерживающего слабое связывание систем, является REST.

В комплект Java EE входят функции, которые реализуют и обеспечивают выполнение технических сквозных задач, таких как управление постоянным хранением данных, настройка и кэширование. Особенно много различных технически обоснованных сценариев удается реализовать посредством CDI. Необходимое асинхронное поведение может быть реализовано по-разному. Приложения не должны сами управлять своими потоками или параллелизмом — для этого следует использовать функции Java EE, такие как управляемые контейнером службы исполнителей, асинхронные события CDI и EJB-таймеры.

Концепции и принципы платформы Java EE поддерживают разработку корпоративных приложений с упором на бизнес-логику. Особенно хороши в этом смысле тесная интеграция различных стандартов, инверсия управления, согласованность конфигурации и *принцип невмешательства*. Инженеры должны стремиться поддерживать высокое качество кода не только путем рефакторинга на уровне кода, но также за счет совершенствования бизнес-логики и создания *универсального языка*, которым будет пользоваться вся команда. Улучшение качества кода, а также достижение соответствия модели предметной области происходят итерационно.

Таким образом, технология должна поддерживать изменения в модели и коде, а также не накладывать слишком много ограничений на используемые решения. В Java EE это реализовано благодаря минимальному влиянию фреймворка на бизнес-код и возможности слабого связывания функций. Участники проекта должны знать, что рефакторинг в сочетании с автоматическим тестированием является необходимым условием разработки высококачественного программного обеспечения.

В следующей главе будет показано, какие еще особенности делают Java EE современной платформой, подходящей для разработки корпоративных приложений. Мы рассмотрим рекомендованные варианты внедрения и узнаем, как заложить основу продуктивных рабочих процессов.

4 Облегченная Java EE

Облегченная Java EE. Неужели это возможно? Раньше приложения J2EE и особенно серверы приложений считались тяжеловесной и громоздкой технологией, и во многом заслуженно. Их API были довольно неудобными в использовании. Требовалась серьезная настройка через XML, что в конечном счете привело к появлению *XDoclet* — инструмента, применяемого для генерации XML на основе метаинформации, помещенной в комментарии JavaDoc. Серверы приложений также были громоздкими в работе, особенно в отношении времени запуска и развертывания.

Однако после того, как название продукта поменялось на Java EE, и особенно с выходом версии 6, все это потеряло актуальность. Появились аннотации, первоначально возникшие из тегов JavaDoc, предназначенных для XDoclet. Появилось и еще много нового, что позволило повысить производительность и сделать работу программистов более эффективной.

В этой главе будут рассмотрены следующие темы.

- Что упрощает технологию.
- Почему стандарты Java EE помогают сократить объем работы.
- Как выбрать зависимости проекта и форматы архива.
- Преимущества корпоративных приложений с нулевой зависимостью.
- Современные серверы приложений Java EE.
- Подход «одно приложение — один сервер приложений».

Облегченная технология корпоративной разработки

Что делает технологию *облегченной*? И насколько облегченной, продуктивной и современной является Java EE в эпоху контейнеров и облаков? Одними из самых важных показателей такой технологии являются обеспечиваемые ею производительность и эффективность. Время, затраченное командой программистов,

важно и дорого, и чем меньше его расходуется на технические задачи, тем лучше. К последним относятся разработка связующего кода, сборка проектов, написание и выполнение тестов, а также развертывание программного обеспечения в локальной и удаленной среде. В идеале инженеры должны тратить как можно больше времени на реализацию бизнес-функций, приносящих доход.

Поэтому технология, за исключением реализации бизнес-сценариев, не должна требовать больших дополнительных расходов. Разумеется, необходимо решать сквозные технические задачи, но их следует свести к минимуму. В предыдущей главе мы увидели, как Java EE позволяет программистам эффективно реализовать бизнес-сценарии. Подобным образом следует максимально сократить время, необходимое на сборку и развертывание артефактов.

В этой и следующей главах будет показано, как Java EE позволяет строить эффективные рабочие процессы.

Зачем нужны стандарты Java EE

В числе главных принципов Java EE — предоставление эффективного корпоративного API. Как мы знаем из предыдущей главы, одно из самых больших преимуществ Java EE — возможность интегрировать различные стандарты таким образом, что программисту нет необходимости их настраивать. Обобщающая спецификация Java EE требует, чтобы разные стандарты хорошо сочетались. Контейнер предприятия должен соответствовать этому требованию. Инженеры-программисты разрабатывают соответствующие API сценарии, перекладывая всю *сложную интеграционную работу* на сервер приложений.

В соответствии с соглашением о конфигурации задействование различных интегрированных стандартов, которые являются частью обобщающей спецификации, не требует начальной настройки. Как вы видели ранее, технологии, основанные на разных стандартах, в Java EE хорошо совместимы друг с другом. Мы рассматривали примеры этого: использование JSON-B для автоматической привязки объектов к JSON в ресурсах JAX-RS; интеграция Bean Validation в JAX-RS и последующих HTTP-ответов с помощью всего одной аннотации; внедрение управляемых компонентов в экземпляры, определенные другими стандартами: валидаторы Bean Validation или адаптеры типа JSON-B; управление техническими транзакциями, охватывающими операции базы данных JPA, в EJB.

Есть ли альтернатива применению обобщающей спецификации, которая охватывает различные технологии многократного использования? Есть: конкретные фреймворки разных производителей со сторонними зависимостями, которые необходимо объединить с тем, что разработано программистами специально для этого приложения. Одно из самых больших преимуществ API Java EE состоит в том, что у программистов под рукой оказывается целый ряд готовых технологий. Это обеспечивает эффективную интеграцию и экономит время, позволяя им сосредоточиться на реализации бизнес-логики.

Соглашения о конфигурации

В соответствии с соглашением о конфигурации корпоративные приложения могут разрабатываться так, что исключается необходимость их начальной настройки. API обеспечивают стандартное поведение, которое соответствует большинству сценариев использования. Инженеру необходимо приложить дополнительные усилия только в том случае, если этих возможностей недостаточно.

Это означает, что в современном мире для настройки корпоративных проектов требуется минимальное изменение конфигурации. Времена обширных XML-конфигураций прошли. В особенности это касается приложений, не имеющих клиентского веб-интерфейса, — в них объем XML-файлов сводится к минимуму.

Начнем с простого примера приложения, предоставляющего конечные точки REST для доступа к базам данных и внешним системам. Конечные точки REST интегрированы посредством технологии JAX-RS, в которой используются сервлеты для обработки запросов. Сервлеты традиционно настраиваются с помощью файла дескриптора развертывания `web.xml`, который находится в `WEB-INF`. Однако JAX-RS отправляет ярлык для создания подкласса `Application`, снабженного аннотацией `@ApplicationPath`, как показано в предыдущей главе, в результате чего сервлет приложения JAX-RS регистрируется по указанному пути. При запуске проект проверяется на наличие связанных классов JAX-RS, таких как ресурсы и поставщики. После запуска приложения конечные точки REST готовы к обработке запросов, даже если не был предоставлен файл `web.xml`.

Управляемые компоненты традиционно настраивались с помощью конфигурационного файла `beans.xml`. В приложениях веб-архивов этот файл также находится в разделе `WEB-INF`. В настоящее время он используется в основном для описания режима обнаружения управляемых компонентов, то есть в нем указано, какие CDI-компоненты должны применяться по умолчанию. Рекомендуется присвоить параметру `bean-discovery-mode` значение `all` (все компоненты), а не только `annotated` (аннотированные). Файл `beans.xml` может переопределять любые CDI, такие как перехватчики, альтернативы, декораторы и т. п. Согласно спецификации CDI в простейшем случае этот файл может быть пустым.

Блоки управления данными JPA настраиваются с помощью файла `persistence.xml` в разделе `META-INF`. Как было показано ранее, он содержит определения источников данных, используемых в приложении. Преобразование JPA-компонентов в таблицы базы данных настраивается посредством аннотаций в классах модели предметной области. Это позволяет хранить все задачи по настройке в одном месте и свести к минимуму применение XML.

Для большинства корпоративных приложений, не имеющих клиентского веб-интерфейса, такой конфигурации вполне достаточно. Клиентские технологии наподобие JSF обычно настраиваются с помощью файлов `web.xml` и `faces-config.xml` либо, если требуется, с помощью дополнительных файлов, характерных для данной реализации.

Раньше файлы конфигурации, специфичные для поставщика, такие как `jboss-web.xml` или `glassfish-web.xml`, встречались довольно часто. В современном мире

Java EE большинство приложений уже не нуждаются в этих обходных решениях. Для того чтобы обеспечить переносимость, настоятельно рекомендуется реализовать функции, используя стандартные API, и только в том случае, если это невозможно или требует чрезмерных усилий, прибегать к функциям, специфичным для поставщика. Как показывает опыт работы с устаревшими проектами, такой подход позволяет создать гораздо более управляемую ситуацию. В отличие от характерных для производителя функций стандарты Java EE гарантируют, что в дальнейшем приложение будет работать.

При запуске приложения контейнер проверяет доступные классы, соответствующие аннотациям и известным типам. Так обнаруживаются и нужным образом настраиваются управляемые компоненты, ресурсы, сущности, расширения и сквозные задачи. Для программистов такой механизм очень удобен. Им не приходится явно указывать требуемые классы в файлах конфигурации, сервер сам их обнаружит — отличная реализация инверсии управления.

Управление зависимостями в проектах Java EE

Управление зависимостями в корпоративном проекте нацелено на зависимости, добавляемые поверх JDK. Сюда относятся зависимости, которые нужны во время компиляции, тестирования и в процессе выполнения приложения. В корпоративном проекте Java требуются Java EE API с предоставленной областью зависимости. Поскольку API уже есть на сервере приложений, их не нужно включать в упакованный архив приложения. Таким образом, предоставленный Java EE API не влияет на размер пакета.

Реальные корпоративные проекты обычно включают в себя другие зависимости. Типичными примерами зависимостей от сторонних разработчиков являются фреймворки протоколирования, такие как *Slf4j*, *Log4j* или *Logback*, фреймворки преобразования JSON, такие как *Jackson*, и библиотеки общего назначения, например *Apache Commons*. Используя их, следует учитывать ряд моментов.

Прежде всего, сторонние зависимости обычно не предоставляются сервером приложений и поэтому увеличивают размер артефакта. Это некритично, но приводит к определенным последствиям, как будет показано позже. Чем больше зависимостей добавляется в артефакт, тем дольше будет выполняться сборка. Системы сборки должны копировать зависимости, которые могут быть довольно большими, в артефакт каждый раз при создании проекта. В главе 6 мы увидим, что сборка проекта должна занимать как можно меньше времени. Каждая зависимость, добавленная в пакет, увеличивает время сборки.

Еще бóльшую проблему представляют собой возможные конфликты зависимостей и их версий. К ним относятся упакованные зависимости, транзитивные зависимости, а также библиотеки, которые уже существуют на сервере приложений. Например, фреймворки регистрации часто уже есть в папке классов контейнера, возможно, в другой версии. Использование различных версий может порождать проблемы с библиотеками. Самой большой из них, как показывает опыт, являются неявные зависимости, которые добавляются транзитивно.

Помимо технических причин, есть и другие особенности, которые следует учесть, прежде чем легкомысленно вводить новые зависимости в программный проект. Например, при разработке коммерческого программного продукта проблемой могут стать лицензии на зависимости. Кроме разрешения на использование определенных зависимостей, нужно, чтобы задействованные лицензии были совместимы между собой при распространении пакета программного обеспечения.

Простейший способ выполнить все условия лицензирования — избегать любых зависимостей, которые не служат достижению коммерческих целей. Аналогичные соображения существуют и относительно безопасности, особенно для тех программных продуктов, которые разрабатываются для областей с высокими требованиями к ней.

Однажды я участвовал в разработке спасательного приложения, обновлявшего версии фреймворков в корпоративном проекте, в который входило множество зависимостей сборки. С учетом всех сторонних зависимостей в процессе выполнения проект в конечном итоге запускал *все* известные фреймворки протоколирования, а также все фреймворки преобразования в JSON, что приводило к многочисленным конфликтам версий и несоответствию зависимостей. Это было до появления JSON-B и JSON-P. Время разработки было потрачено главным образом на настройку сборки проекта, распутывание и исключение транзитивных зависимостей в артефакте проекта. Это типичная проблема при использовании сторонних библиотек. Для сохранения кода проекта придется потратить время и усилия: настроить сборку проекта и распутать зависимости, особенно если они вводят много транзитивных функций.

Управляя зависимостями сборки, инженеры сосредотачиваются на деталях, не влияющих на бизнес-сценарии. Возникает вопрос: стоит ли стараться сохранить несколько строк кода, если при этом вводятся дополнительные зависимости? Опыт показывает, что выбор между дублированием и малым объемом, как в проектах без зависимостей, слишком часто склоняется в сторону предотвращения дублирования. Ярким примером этого являются проекты, включающие в себя всю библиотеку Apache Commons, чтобы применять функции, которые могли бы быть реализованы всего несколькими строками кода.

Хорошим стилем считается не изобретать велосипед, разрабатывая собственные версии функций, а использовать уже существующие. Однако при этом нужно учитывать возможные последствия. Как показывает опыт, введенные зависимости довольно часто игнорируют и применяют очень ограниченно.

Когда инженеры проверяют качество кода, например используя инструменты анализа кода, им следует учитывать также соотношение зависимостей и кода проекта, которые решают задачи бизнеса и вспомогательные задачи. Существует простой метод для принятия решения по поводу зависимостей. Прежде чем вводить стороннюю зависимость, ответьте на следующие вопросы.

- Повысится ли в результате ценность приложения для бизнеса?
- Сколько строк кода проекта она сэкономит?
- Насколько велико будет ее влияние на размер артефакта?

Например, представьте, что часть сценариев в приложении по производству автомобилей служит для обмена данными с конкретным заводским программным обеспечением посредством API Java собственной разработки. Очевидно, что для реализации бизнес-логики эта коммуникация имеет решающее значение, так что стоит включить эту зависимость в проект. А вот добавление еще одного фреймворка протоколирования, напротив, вряд ли повысить ценность приложения для бизнеса. Подробнее о проблемах традиционного протоколирования читайте в главе 9.

Для того чтобы не увеличивать размер сборки, можно установить критически важные зависимости на сервере приложений и объявить их в сборке проекта как предоставляемые извне.

В главе 1 мы рассмотрели трудности, возникающие при совместном использовании таких бизнес-зависимостей, как общие модели. В идеале приложения должны быть максимально самодостаточны. В главе 8 подробно обсудим автономные системы и причины создания архитектур, которые не задействуют совместный доступ к чему бы то ни было.

Однако с техническими зависимостями ситуация другая. API Java EE включает в себя технологию, которая требуется для большинства корпоративных приложений. В идеале инженеры разрабатывают приложение Java EE с нулевой зависимостью, упакованное в виде «тонкого» артефакта развертывания, содержащего только классы, относящиеся к приложению. Если некоторые сценарии использования требуют сторонних зависимостей, то они могут быть установлены в контейнере. Цель состоит в том, чтобы получить артефакт развертывания малого объема.

Это означает, что в готовый код приложения включены только предоставляемые зависимости, в идеале только Java EE API. Однако тестовые зависимости — это совсем другое дело: для тестирования программного обеспечения требуются некоторые дополнительные технологии. Подробнее о зависимостях, необходимых для тестирования, читайте в главе 7.

Облегченный способ упаковки приложений

В приложениях с нулевой зависимостью решаются многие проблемы, связанные со сборкой проектов. Поскольку нет сторонних зависимостей, то нет и необходимости управлять ими, чтобы обеспечить соответствие версий и избежать конфликтов.

Что еще упрощается благодаря такому подходу? Сборка проекта независимо от того, используется Gradle или Maven, всегда проходит быстрее, если не нужно ничего добавлять в артефакт. Размер создаваемых пакетов напрямую влияет на время сборки. В приложения с нулевой зависимостью включены только скомпилированные классы, то есть только фактическая бизнес-логика. Поэтому время сборки такого приложения будет минимальным. Все время сборки будет расходоваться только на компиляцию классов проекта, выполнение тестовых примеров

и упаковку классов в «тонкий» артефакт развертывания. Такая сборка занимает несколько секунд — да, именно секунд. Как правило, если сборка проекта длится больше десяти секунд, ее следует пересмотреть.

Разумеется, это правило оказывает определенное давление на проекты. Оно, естественно, требует избегать включения в них больших зависимостей и реализаций — они должны предоставляться сервером приложений. Еще одним фактором, замедляющим сборку, обычно является время выполнения тестов. О том, как создавать эффективные тесты, вы узнаете в главе 7.

Быстрые сборки — одно из преимуществ создания приложений с нулевой зависимостью. Другим является быстрая передача артефактов. Готовые артефакты в виде WAR- или JAR-файлов обычно содержатся в хранилище артефактов, таком как *Sonatype Nexus* или *JFrog Artifactory*, в состоянии, готовом к использованию. Передача этих артефактов по сети значительно ускоряется, если речь идет всего о нескольких килобайтах данных. Это касается всех видов развертывания артефакта. Независимо от того, куда передаются архивы сборок, чем меньше их размер, тем лучше, особенно если рабочие процессы выполняются часто, как в случае непрерывной поставки.

К пересмотру и исключению всего того, что не имеет ценности для бизнеса, относится и изменение способа упаковки приложений. Традиционно корпоративные приложения поставляются в виде EAR-файлов. В их состав входят веб-архив, WAR-файл и один или несколько корпоративных JAR-файлов. Корпоративные JAR-архивы содержат бизнес-логику, обычно в виде EJB-компонентов. Веб-архив содержал веб-сервисы и клиентские технологии для обмена данными с бизнес-логикой посредством локальных или удаленных EJB-компонентов. Однако такое разделение необязательно, поскольку все компоненты поставляются в общем экземпляре сервера.

Сейчас уже не нужно упаковывать разные технические блоки в отдельные субархивы. Вся бизнес-логика вместе с веб-сервисами и сквозным функционалом упаковывается в один WAR-файл. Это значительно упрощает установку проекта, а также процедуру сборки. Теперь не приходится упаковывать приложение в виде нескольких уровней иерархии, чтобы затем снова распаковать его в один экземпляр сервера. Наилучшей реализацией «тонких» артефактов являются WAR-файлы, содержащие необходимый бизнес-код, развертываемый в виде контейнера. По этой причине развертывание «тонких» WAR-файлов выполняется быстрее, чем соответствующих EAR-файлов.

На рис. 4.1 показан артефакт «тонкого» веб-приложения, содержащего типичные компоненты.

Артефакт развертывания содержит только классы, необходимые для реализации бизнес-сценария, без реализаций, специфичных для данной технологии, и представлен в минимальной конфигурации. В частности, в артефакт не включаются JAR-файлы библиотек.

Архитектура платформы Java EE приветствует легкие артефакты. Это связано с тем, что в данной платформе разделены API и реализации. Разработчики программируют, только используя API. сервер приложений реализует этот API.

Это позволяет поставлять лишь бизнес-логику, создавая легкие артефакты и включая в них только некоторые компоненты приложения. Помимо очевидных преимуществ, таких как отсутствие конфликтов зависимостей и создание решений, не зависящих от поставщика, такой подход обеспечивает также быстрое развертывание. Чем меньше содержимое артефакта, тем быстрее происходит его распаковка на стороне контейнера. Поэтому настоятельно рекомендуется упаковывать корпоративные приложения в один WAR-файл.

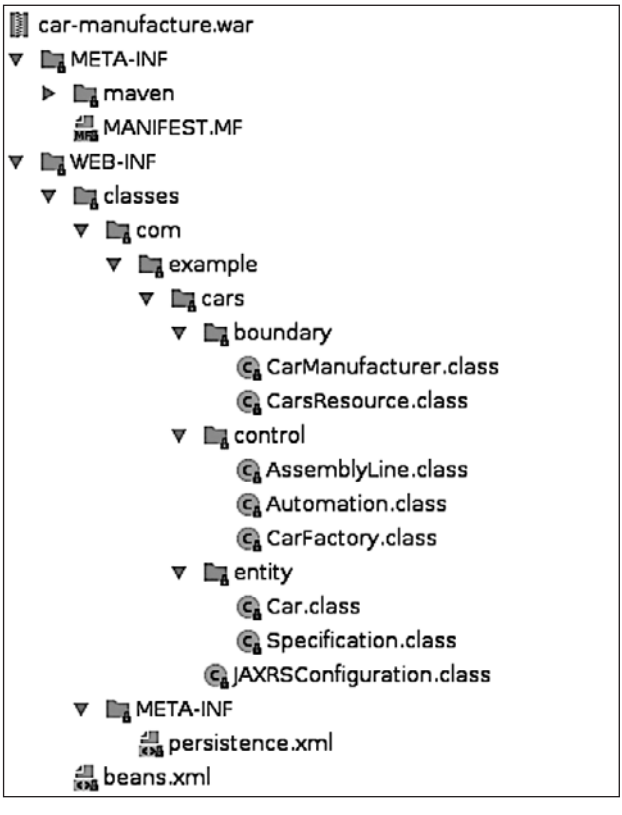

Рис. 4.1

В 2016 году мы наблюдали растущий интерес к поставке корпоративных приложений в виде «толстых» JAR-файлов, то есть к поставке приложений вместе с реализацией. Развертывание «толстых» артефактов обычно применяется в таких корпоративных фреймворках, как Spring Framework. Причиной такого подхода является то, что версии нужных зависимостей и фреймворков четко указаны и поставляются вместе с бизнес-логикой. «Толстые» артефакты развертывания могут создаваться в виде больших WAR-файлов, которые развертываются в контейнере сервлета, или больших исполняемых JAR-файлов, запускаемых автономно. Таким образом, приложение Java EE, упакованное в виде исполняемого

«толстого» JAR-файла, помимо самого приложения, содержит корпоративный контейнер. Однако, как уже отмечалось, если включить в артефакт сторонние зависимости, то время сборки, поставки и развертывания значительно возрастает.

Опыт показывает, что явная поставка всей корпоративной реализации вместе с приложением в большинстве случаев обусловлена не техническими характеристиками, а политикой предприятия. Операционные среды внутри компаний негибки по отношению к серверу приложений и инсталляциям Java, особенно в части обновлений версий, и в некоторых случаях это вынуждает программистов искать обходные пути. Корпоративные приложения, построенные на базе более новой технологии, нельзя развернуть на старых инсталляциях сервера. Иногда с точки зрения политики предприятия проще полностью проигнорировать существующие инсталляции и явно запустить автономный JAR-файл, требующий только определенной версии Java. Но хотя эти решения, безусловно, в каком-то смысле оправданны, технически более разумно было бы упаковывать приложения в тонкие артефакты развертывания. Любопытно, что, как мы увидим в следующей главе, программное обеспечение для поставки в контейнерах Linux имеет преимущества обоих подходов.

Существует еще один интересный подход, который позволяет поставлять все приложение в виде исполняемого пакета и поддерживать быстрые рабочие процессы «тонкого» развертывания. Некоторые поставщики серверов приложений предоставляют решение для поставки специального контейнера приложений в виде исполняемого JAR-файла, который развертывает «тонкое» приложение как дополнительный аргумент во время запуска. Таким образом, весь пакет с обоими артефактами включает в себя бизнес-логику и реализацию и запускается как отдельное приложение. Приложение по-прежнему отделено от среды выполнения и упаковано как «тонкий» артефакт в виде так называемого *пологого* JAR- или WAR-файла. Такой подход особенно эффективен, если нужно обеспечить гибкость без использования контейнеров Linux.

В заключение отмечу: настоятельно рекомендуется создавать «тонкие» артефакты развертывания, в идеале в виде «тонких» WAR-файлов. Если этого не допускает политика предприятия, то разумным решением проблемы будут полые JAR-файлы. Однако, как мы увидим в следующей главе, контейнерные технологии, такие как Docker, не требуют исполняемых JAR-файлов и обеспечивают те же преимущества.

Серверы приложений Java EE

Что еще упрощает технологию создания корпоративных приложений, кроме продуктивности API? Как насчет среды выполнения, корпоративного контейнера?

Прежде программисты часто жаловались на то, что сервер приложений J2EE слишком медленный, слишком неудобный и громоздкий. Размеры инсталляции и потребление памяти были довольно высокими. Как правило, многие приложения выполнялись на экземпляре сервера и одновременно перераспределялись,

каждое в отдельности. Такой подход иногда создавал дополнительные сложности, в частности проблемы с иерархией классов.

Современные серверы приложений далеко ушли от этого. Большинство из них значительно оптимизированы для сокращения времени запуска и развертывания. В особенности это касается внутренних серверных модулей, таких как *Open Service Gateway Initiative (OSGi)*, который позволил обойтись без обязательной поддержки полного API Java EE, так что необходимые модули теперь загружаются по требованию, а это значительно ускоряет работу. Что касается использования ресурсов, то серверы приложений также значительно улучшились по сравнению с тем, какими были раньше. Современный контейнер потребляет меньше памяти, чем работающий экземпляр браузера на настольном компьютере. Например, экземпляр *Apache TomEE* запускается за одну секунду, потребляет менее 40 Мбайт дискового пространства и менее 30 Мбайт памяти.

Расход производительности на управляемые компоненты также незначителен. Фактически, по сравнению с управляемыми CDI-компонентами и другими структурами, такими как Spring Framework, EJB-компоненты без сохранения состояния показывают наилучшие результаты. Это связано с тем, что компоненты сессий без сохранения состояния заносятся в накопитель и повторно применяются после вызова соответствующих бизнес-методов.

Кроме того, серверы приложений управляют накопителями соединений и потоков и позволяют инженерам собирать статистику и данные о производительности без дополнительной настройки. Контейнер обеспечивает мониторинг этих показателей. У инженеров DevOps есть возможность использовать эти данные напрямую, без создания специальных параметров.

К тому же, как мы узнали из предыдущей главы, серверы приложений также управляют экземплярами компонентов и жизненными циклами, ресурсами и транзакциями базы данных.

Все дело здесь в контейнере приложения. Он выполняет всю работу по запуску корпоративного приложения — инженеры-программисты отвечают только за бизнес-логику. Контейнер обеспечивает необходимые ресурсы и управляет ими, он действует в рамках стандартов предоставления данных для развернутых приложений. Поскольку поставщики прилагают большие усилия для оптимизации необходимых технологий, объем нерационально используемых ресурсов сводится к минимуму.

Размеры инсталляций серверов приложений все еще несколько больше, чем у других корпоративных фреймворков. На момент написания этой книги поставщики стремились предоставить по требованию в соответствии с потребностями приложения среды выполнения меньшего размера. Инициатива *MicroProfile* объединила нескольких поставщиков серверов приложений, которые создают корпоративные профили, дополняющие обобщающую спецификацию Java EE. Эти профили также собираются из стандартов Java EE. С точки зрения программистов, это очень интересный подход, поскольку он не требует каких-либо изменений на стороне приложения. Среда выполнения, то есть набор включенных стандартов, будет соответствовать потребностям приложения для реализации его бизнес-логики.

Одно приложение — один сервер приложений

Традиционно из-за больших размеров инсталляции и длительного времени запуска серверы приложений использовались для развертывания нескольких единиц, если не десятков, корпоративных приложений. Экземпляром сервера пользовались одновременно несколько команд, иногда вся компания. Это приводило к некоторой негибкости, как в случае общих моделей приложения. Отдел компании не мог установить более новую версию JDK или сервера, перезапустить или перенастроить сервер приложений, не согласовав это с другими командами. Это, естественно, замедляло процессы, уменьшало их продуктивность и усложняло непрерывную поставку.

Что касается методов работы в команде, а также жизненных циклов проектов и приложений, то самым простым вариантом в этом случае является развертывание приложения на отдельном сервере приложений. При этом команда DevOps полностью контролирует версию, конфигурацию и жизненный цикл своего приложения. Такой подход позволяет упростить процессы и избежать возможных проблем, таких как конфликты с другими командами компании и используемыми технологиями. Также устраняются проблемы иерархической загрузки классов, которые могли бы появиться при установке нескольких приложений.

Определенно, сервер приложений — это конструкция, предназначенная только для одного приложения. Однако, как мы уже знаем, размеры инсталляций серверов приложений заметно уменьшились по сравнению с прошлым. Кроме этого, программисты должны уделять больше внимания размерам артефактов развертывания, поскольку они являются подвижными частями процесса разработки. При непрерывной поставке приложение потенциально собирается и упаковывается много раз в день. Чем больше времени тратится на сборку и передачу артефактов проекта, тем длиннее цикл производства. В сумме все сборки, выполняемые в течение дня, могут занимать значительное время. Сервер приложений устанавливается и развертывается гораздо реже. Поэтому рекомендуется развернуть приложение на одном выделенном Java EE-сервере, как показано на рис. 4.2.

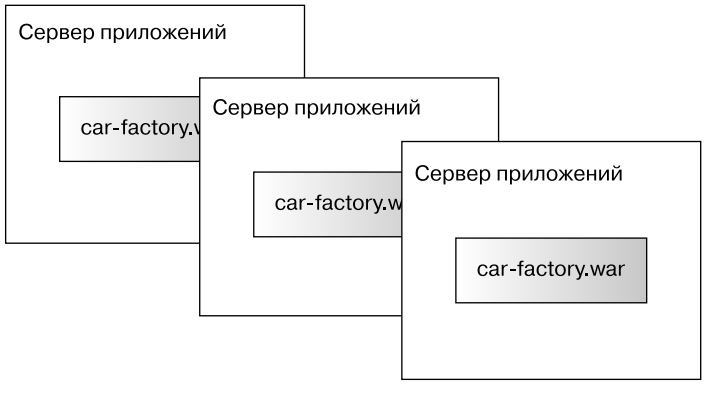

Рис. 4.2

В следующей главе мы увидим, как контейнерные технологии, подобные Docker, поддерживают такой подход. Поставка приложения, включая весь стек вплоть до операционной системы в качестве контейнера, поощряет подход «одно приложение — один сервер приложений».

Резюме

Полная интеграция нескольких стандартов Java EE в сочетании с соглашением о конфигурации сводит к минимуму количество стереотипного кода, который приходится писать программистам. Таким образом, конфигурация современных корпоративных приложений минимизирована. Особенно полезны соглашения по умолчанию, которые подходят для большинства корпоративных приложений, и возможность переопределения конфигурации только в случае необходимости, что значительно повышает продуктивность работы программиста.

В корпоративных приложениях следует свести к минимуму зависимости, в идеале ограничиться только предоставляемыми Java EE API. Сторонние зависимости следует добавлять только в том случае, если они необходимы для бизнеса, а не для решения технических задач.

В соответствии с принципом нулевой зависимости приложения Java EE должны быть упакованы в виде «тонких» WAR-файлов. Это положительно влияет на время, затрачиваемое на сборку и публикацию приложения.

Современные приложения Java EE — уже не прежние тяжеловесные продукты на базе J2EE. Они запускаются и развертываются быстро и стремятся уменьшить нагрузку на память. Серверы приложений могут быть не самыми легкими в процессе выполнения, но они обеспечивают корпоративным приложениям значительные преимущества, такие как интеграция технологий и управление жизненными циклами, соединениями, транзакциями и потоками, которые иначе пришлось бы реализовывать в самом приложении.

Для упрощения жизненного цикла и развертывания приложений рекомендуется использовать для каждого из них отдельный сервер приложений. Это устраняет сразу несколько потенциальных проблем и прекрасно вписывается в современный мир контейнерных технологий. В следующей главе я покажу вам этот современный мир в эпоху облачных платформ и познакомлю с контейнерными технологиями. Вы увидите, как Java EE вписывается в эту картину.

5 Java EE в контейнерных и облачных средах

За последние годы значительно вырос интерес к контейнерным и облачным технологиям. Подавляющее большинство компаний, создающих программное обеспечение, по крайней мере рассматривают переход своих систем на эти современные технологии. Во всех моих последних проектах эти технологии были предметом обсуждения. В частности, внедрение технологий управления контейнерами существенно влияет на стиль работы приложений.

В чем преимущества контейнерных технологий? И почему компаниям стоит подумать об облаке? Похоже, что об этих принципах часто говорят лишь ради красного словца или в надежде на волшебное средство, которое устранит все проблемы. В данной главе мы исследуем причины применения этих технологий. А также узнаем, готова ли платформа Java EE к вступлению в новый мир и что в ней сделано для этого.

В этой главе будут рассмотрены следующие вопросы.

- Как инфраструктура в виде кода поддерживает операции.
- Контейнерные технологии и управление ими.
- Почему именно Java EE подходит для этих технологий.
- Облачные платформы и причины их применения.
- Двенадцатифакторные облачные корпоративные приложения.

Цели и обоснование использования

Каковы причины использования контейнеров, систем управления контейнерами и облачных сред? Почему эта область получила такой мощный толчок к развитию?

Традиционно развертывание корпоративных приложений работало примерно так: разработчики приложений реализовывали некоторую бизнес-логику и собирали приложение в виде упакованного артефакта. Он развертывался вручную

на сервере приложений, который также управлялся вручную. Во время развертывания или реконфигурации сервера приложение обычно простаивало.

Естественно, такой подход был связан со значительными рисками. Решение задач вручную человеком сопряжено с ошибками, к тому же нет гарантии, что они всегда будут выполняться одинаково. Люди довольно плохо выполняют автоматическую, повторяющуюся работу. Такие процессы, как установка серверов приложений, операционных систем и серверов в целом, требуют четкой документации, особенно для того, чтобы обеспечить воспроизводимость в будущем.

Раньше на эксплуатационные задачи обычно оформлялись заказы с использованием билетной системы, и эти задачи выполнялись вручную. Таким образом, при установке и настройке серверов существовал риск того, что система перейдет в невоспроизводимое состояние. Для построения новой среды, идентичной существующей, требовалось вручную провести множество исследований.

Необходимо было автоматизировать оперативные задачи и сделать их воспроизводимыми. Установка нового сервера, операционной системы или среды выполнения всегда должна проходить одинаково. Автоматизация процессов не только ускоряет их, но и обеспечивает прозрачность, так как при этом видно, какие именно операции были реализованы. Переустановка среды должна обеспечивать точно такую же систему, как и раньше, включая всю конфигурацию и настройку.

Все сказанное относится также к развертыванию и настройке приложения. Вместо ручной сборки и поставки приложений *серверы непрерывной интеграции (Continuous Integration, CI)* обеспечивают автоматическую надежную и воспроизводимую сборку программного обеспечения. Серверы CI выступают в качестве *истины в последней инстанции* при сборке программного обеспечения. Создаваемые с их помощью артефакты развертываются во всех задействованных средах. Программный артефакт собирается один раз на сервере непрерывной интеграции, проходит интеграционные и сквозные тесты, пока не будет получен готовый продукт. Таким образом, один и тот же двоичный код приложения, развертываемый в рабочих средах, предварительно надежно тестируется.

Еще одна очень важная особенность заключается в явном указании используемых версий программного обеспечения. Это относится ко всем программным зависимостям, от сервера приложений и среды Java до операционной системы и ее двоичных файлов. При восстановлении или переустановке программного обеспечения каждый раз должно обеспечиваться одно и то же состояние системы. Программные зависимости — сложная тема, они могут стать причиной разнообразных ошибок. Приложения тестируются для правильной работы в определенных средах с конкретными конфигурациями и зависимостями. Для того чтобы гарантировать, что приложение будет работать должным образом, оно поставляется именно в той конфигурации, которая была проверена ранее.

Это также подразумевает, что тестовые и промежуточные среды, применяемые для проверки поведения приложения, должны быть максимально похожими на эксплуатационные. Теоретически это ограничение звучит разумно. Однако, как показывает опыт, среда разработки может значительно отличаться от среды

эксплуатации в смысле используемых версий программного обеспечения, конфигурации сети, баз данных, внешних систем, количества экземпляров сервера и т. д. Для того чтобы правильно протестировать приложения, эти различия следует свести к минимуму. В разделе «Контейнеры» будет показано, как в этом могут помочь контейнерные технологии.

Инфраструктура как код

Логичным решением, позволяющим создать воспроизводимую среду, является использование *инфраструктуры как кода (infrastructure as code, IaC)*. Смысл его в том, чтобы все необходимые операции, конфигурации и версии были явно определены в виде кода. Эти определения в виде кода задействуются непосредственно для настройки инфраструктуры. Инфраструктура как код может быть реализована в процедурной форме, такой как сценарии, или в виде объявлений. Последний вариант означает описание состояния, которого надо достичь, и выполняется с применением дополнительного инструментария. Независимо от того, какой вариант выбран, вся среда должна быть описана как код, реализуемый автоматически надежным и воспроизводимым способом, всегда обеспечивающим одни и те же результаты.

В любом случае это подразумевает, что ручные операции сводятся к минимуму. Простейшей формой инфраструктуры как кода являются сценарии оболочки. Они от начала до конца должны выполняться без участия человека. Это касается всех решений IaC.

Разумеется, ответственность за установку и настройку среды переходит от эксплуатационной группы к программистам. Поскольку команда программистов устанавливает определенные требования к необходимой операционной среде, следует наладить сотрудничество между всеми командами инженеров. Именно такая идея лежит в основе движения DevOps. В прошлом слишком часто происходило так: разработчики приложений создавали программное обеспечение, передавали его вместе с остальными обязанностями эксплуатационщикам и в дальнейшем никак не участвовали в процессе. Потенциальные ошибки на этапе эксплуатации в первую очередь были проблемой эксплуатационной группы. Все это не только приводило к напряженным отношениям между командами инженеров, но и в конечном счете снижало качество продукта. Тогда как общая цель должна заключаться в создании высококачественного программного обеспечения, выполняющего свои задачи.

Эта цель требует от разработчиков приложений ответственности. Когда вся необходимая инфраструктура, конфигурация и программное обеспечение определены как код, все команды естественным образом вместе движутся к намеченной цели. Движение DevOps направлено на усиление ответственности команды разработчиков программного обеспечения за продукт в целом. Представление инфраструктуры как кода — это необходимое предварительное условие, которое позволяет повысить воспроизводимость, автоматизацию и в целом качество программного обеспечения.

В разделах «Контейнеры» и «Фреймворки управления контейнерами» будет показано, как принцип IaC реализован в существующих технологиях.

Стабильность и готовность к эксплуатации

Практика непрерывной поставки включает в себя то, что необходимо сделать для повышения качества и ценности программного обеспечения. Это, в частности, стабильность приложения. Перенастройка и повторное развертывание программного обеспечения не должны приводить к простоям. Новые функции и исправления не должны устанавливаться только во время обслуживания. В идеале корпоративное программное обеспечение может непрерывно улучшаться и совершенствоваться.

Добиться *нулевых простоев* можно, приложив определенные усилия. Для того чтобы избежать периодов, когда приложение недоступно, по крайней мере один экземпляр программного обеспечения должен быть доступен в любой момент времени. Распределитель нагрузки или прокси-сервер заранее должен направлять трафик на доступный экземпляр приложения. Эта технология применяется при «сине-зеленом» развертывании, как показано на рис. 5.1.

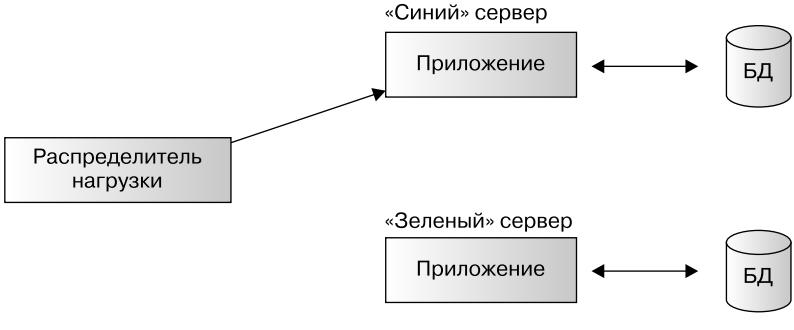

Рис. 5.1

Экземпляры приложений вместе с их базами данных реплицируются и проксируются с помощью распределителя нагрузки. Обычно задействованные приложения представляют собой различные версии программного обеспечения. Переключение трафика с «синего» на «зеленый» сервер и наоборот мгновенно изменяет версию, и простоя не наблюдается. Другие сценарии «сине-зеленого» развертывания могут задействовать несколько экземпляров приложений, настроенных на использование одного и того же экземпляра базы данных.

Этот подход, очевидно, может быть реализован без применения какой-то блестящей новой технологии. Прежде нам встречались «сине-зеленые» развертывания, обеспечивающие нулевое время простоя, которые были реализованы на базе решений собственной разработки. Тем не менее современные технологии поддерживают эти методы, повышая стабильность, качество и готовность к эксплуатации по умолчанию, без особых технических усилий.

Контейнеры

В последние годы появился большой интерес к технологии *контейнеров Linux*. Технически этот подход не новый. Операционные системы Linux, такие как *Solaris*, поддерживающие контейнеры, существуют давным-давно. Тем не менее *Docker* стал прорывом в этой технологии, предоставляя функции для единого способа сборки контейнеров, управления ими и их поставки.

В чем разница между контейнерами и *виртуальными машинами (virtual machines, VM)* и что интересного в контейнерах?

Виртуальные машины действуют как компьютер в компьютере. Они позволяют легко управлять средой извне — например, создавать, запускать, останавливать и распределять виртуальные машины быстро, идеально автоматизированным способом. Если необходимо установить новый сервер, то его копия, или образ требуемого типа, может быть развернута так, что не потребуется каждый раз заново устанавливать программное обеспечение. Для того чтобы упростить резервное копирование текущего состояния, можно сделать снимки рабочей среды.

Контейнеры во многих отношениях подобны виртуальным машинам. Они отделены от хоста, а также от других контейнеров, запускаются в собственной сети со своей файловой системой и, возможно, с отдельными ресурсами. Разница заключается в том, что виртуальные машины работают на уровне абстракции оборудования, эмулируя компьютер, в том числе операционную систему, тогда как контейнеры запускаются непосредственно в ядре хоста. В отличие от других процессов ядра контейнеры отделены от остальной системы и используют функциональные возможности операционной системы. Они управляют собственной файловой системой, поэтому ведут себя как отдельные машины, но их производительность максимальна и не расходуется на поддержку уровня абстракции. Производительность же виртуальных машин естественным образом уменьшается за счет их абстракции. В то время как виртуальные машины очень гибко подходят к выбору операционной системы, контейнеры всегда будут работать на том же ядре и, следовательно, в той же версии, что и операционная система хоста. Таким образом, контейнеры не содержат собственного ядра Linux и их размер может быть сведен к минимуму — до необходимых бинарных файлов.

Контейнерные технологии, такие как Docker, обеспечивают функциональность для единообразных сборки, запуска и распространения контейнеров. В Docker сборка образов контейнеров определена как IaC, что позволяет автоматизировать процесс, повысить его надежность и воспроизводимость. В файлах Docker описаны все операции, необходимые для установки приложения, включая его зависимости, например контейнер приложения и среду Java. Каждый шаг в файле Docker соответствует команде, выполняемой во время сборки образа. При запуске из образа контейнер должен содержать все, что для этого требуется.

Обычно контейнеры содержат один процесс Unix, представляющий собой запущенную службу, например сервер приложений, веб-сервер или базу данных. Если корпоративная система состоит из нескольких работающих серверов, то они запускаются в отдельных контейнерах.

Одним из самых больших преимуществ контейнеров Docker является то, что они используют файловую систему с *копированием при записи*. Каждая операция сборки, а впоследствии каждый запущенный контейнер работает в многоуровневой файловой системе, в которой уровни остаются неизменными и только поверх существующих добавляются новые. Поэтому собранные образы содержат несколько уровней.

Контейнеры, создаваемые из образов, всегда запускаются из одного и того же начального состояния. В ходе работы контейнеров файлы могут изменяться, при этом в файловой системе создаются новые временные уровни, которые будут отброшены после того, как контейнер завершит работу. Таким образом, по умолчанию контейнеры Docker являются рабочими средами без сохранения состояния. Это стимулирует идею воспроизводимости. Каждое постоянное поведение должно быть описано явно.

Многочисленные уровни полезны при повторной сборке и распространении образов. Docker кэширует промежуточные уровни, пересобирает и повторно отправляет только то, что было изменено.

Например, сборка образа может состоять из нескольких этапов. Сначала в нее включаются системные двоичные файлы, затем среда выполнения Java, сервер приложений и, наконец, само приложение. Если в приложение внесены изменения и требуется новая сборка, то выполняется только последний этап, так как предыдущие этапы были закэшированы. То же самое происходит и при передаче образов по сети. Повторно передаются только те уровни, которые были изменены и которых еще нет в реестре получателя.

На рис. 5.2 показаны уровни образа Docker и их индивидуальное распространение.

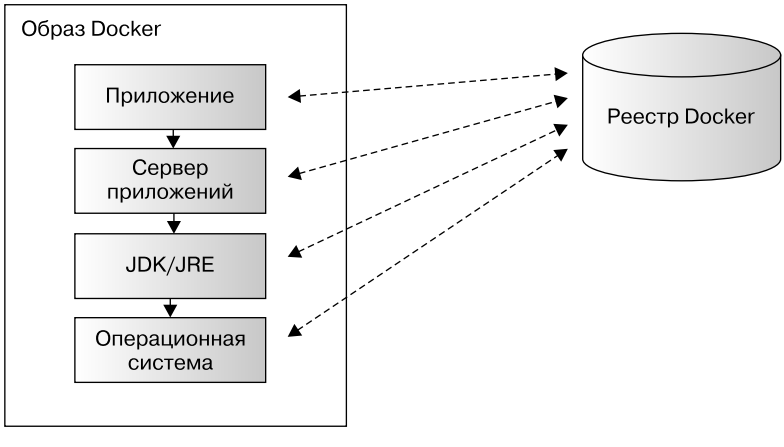

Рис. 5.2

Образы Docker создаются либо с нуля, либо с пустой начальной точки, либо на основе существующего базового образа. Есть множество готовых базовых образов для всех основных дистрибутивов Linux с менеджерами пакетов для типичных

стеков среды, а также для образов на основе Java. Базовые образы — это способ создать сборки на единой основе и обеспечить базовую функциональность для всех получаемых образов. Например, имеет смысл использовать базовый образ, включающий в себя среду выполнения Java. Если его потребуется обновить, например, чтобы исправить проблемы безопасности, то можно будет перестроить все зависимые образы и получить новое содержимое, обновив одну версию базового образа. Как уже отмечалось, сборка программного обеспечения должна обеспечивать повторяемость. Поэтому всегда нужно указывать явные версии артефактов программного обеспечения, таких как образы.

Контейнеры, которые запускаются из ранее созданных образов Docker, нуждаются в доступе к этим образам. Они распространяются с использованием реестров Docker, таких как общедоступные DockerHub или внутренние реестры компании для распространения собственных образов. Локально собранные образы помещаются в эти реестры и извлекаются оттуда в средах, в которых затем будут запускаться новые контейнеры.

Java EE в контейнере

Как оказалось, концепция многоуровневой файловой системы соответствует существующей в Java EE концепции разделения приложения и среды выполнения. «Тонкие» артефакты развертывания содержат только фактическую бизнес-логику — ту часть, которая постоянно изменяется и перестраивается. Эти артефакты развертываются в корпоративном контейнере, который изменяется не так уж часто. Образы контейнера Docker собираются поэтапно, уровень за уровнем. Сборка образа корпоративного приложения включает в себя базовый образ операционной системы, среду выполнения Java, сервер приложений и, наконец, само приложение. Если изменяется только уровень приложения, то только его нужно пересобрать и повторно передать — все остальные уровни изменяются только один раз, а затем кэшируются.

«Тонкие» артефакты развертывания используют преимущества уровней — это позволяет перестраивать и повторно распространять всего несколько килобайт. Таким образом, рекомендуется применять в контейнерах именно приложения с нулевой зависимостью.

Как говорилось в предыдущей главе, имеет смысл развертывать каждое приложение на отдельном сервере приложений. Контейнер реализует только один процесс — в данном случае это сервер приложений, содержащий само приложение. Поэтому сервер приложений должен запускаться в отдельном контейнере, который, в свою очередь, включен в контейнер. И сервер приложений, и само приложение добавляются в образ в процессе сборки. Потенциальная конфигурация, например, источников данных, накопителей или модулей сервера также создается в процессе сборки, обычно добавлением специализированных файлов конфигурации. Поскольку контейнер содержит только одно приложение, то конфигурация этих компонентов больше ни на что не повлияет.

Запущенный из образа контейнер должен содержать все, что требуется для его работы. В частности, в нем должно присутствовать приложение вместе со всей необходимой конфигурацией. Поэтому приложения больше не развертываются в уже запущенном контейнере, а добавляются в него на этапе сборки образа, чтобы быть на месте в момент запуска контейнера. Обычно для этого артефакт развертывания помещают в каталог автоматического развертывания контейнера. Приложение будет развернуто при запуске сконфигурированного сервера приложений.

Образ контейнера собирается только один раз, а затем выполняется во всех средах. В соответствии с изложенной ранее концепцией воспроизводимых артефактов, используемые в рабочей среде артефакты должны быть заранее протестированы. Поэтому в рабочей среде публикуется предварительно верифицированный образ Docker.

Но что делать, если приложения имеют разную конфигурацию в разных средах? Что делать, если нужно наладить коммуникацию с различными внешними системами или базами данных? Чтобы не создавать конфликт между разными средами, по крайней мере следует использовать разные экземпляры базы данных. Приложения, поставляемые в контейнерах, запускаются из одного и того же образа, но иногда все еще нуждаются в некоторых изменениях.

Docker позволяет изменять некоторые характеристики работающих контейнеров: создание сетей, добавление томов, то есть вставку файлов и каталогов, расположенных на хосте Docker, а также создание переменных среды Unix. Различия параметров среды добавляются посредством управления контейнером извне. Образы собираются только один раз для конкретной версии и затем используются и, возможно, модифицируются в разных средах. Это обеспечивает большое преимущество в том смысле, что различия в конфигурации не моделируются внутри приложения, а управляются извне. Как мы увидим в следующих разделах, это верно и для настройки сетей, а также подключения приложений и внешних систем.

Кстати, Linux-контейнеры решают часто возникающую вследствие корпоративной политики проблему поставки приложения вместе с реализацией в едином пакете из соображений гибкости. Поскольку контейнеры включают в себя среду выполнения и все необходимые зависимости, в том числе среду Java, инфраструктура должна обеспечивать только среду выполнения Docker. Все применяемые технологии, включая версии, должны предоставляться группой программистов.

В следующем фрагменте кода показано определение `Dockerfile` для сборки корпоративного приложения `hello-cloud` с получением базового образа *WildFly*:

```
FROM jboss/wildfly:10.0.0.Final

COPY target/hello-cloud.war /opt/jboss/wildfly/standalone/deployments/
```

`Dockerfile` определяет базовый образ `jboss/wildfly` в заданной версии, которая уже содержит среду выполнения Java 8 и сервер приложений WildFly. Он находится в каталоге проекта приложения и ссылается на архив `hello-cloud.war`, который был собран ранее посредством Maven. WAR-файл скопирован в каталог автоматического развертывания `WildFly` и будет доступен в этом месте во время выполнения контейнера. Базовый образ `jboss/wildfly` уже содержит команду

запуска, где указано, как запустить сервер приложений, унаследованный Dockerfile. Поэтому в нем не нужно задавать эту команду еще раз. После сборки Docker готовый образ будет содержать все, что было в базовом образе `jboss/wildfly`, включая приложение hello-cloud. Это соответствует концепции установки сервера приложений WildFly с нуля и добавления WAR-файла в каталог автоматического развертывания. При распространении собранного образа необходимо будет передать только добавленный уровень, содержащий «тонкий» WAR-файл.

Модель развертывания для платформы Java EE соответствует концепции контейнеров. Разделение приложения на уровни корпоративного контейнера позволяет использовать файловые системы с копированием при записи, сводя к минимуму время, затрачиваемое на сборку, распространение и развертывание.

Фреймворки управления контейнерами

Подробнее рассмотрим уровень абстракции, лежащий над контейнерами. Контейнеры включают в себя все необходимое для запуска конкретных сервисов без сохранения состояния, в виде автономных артефактов. Однако контейнеры нуждаются в настройке для правильной работы в сети, чтобы при необходимости обмениваться данными с другими сервисами и запускаться в правильной конфигурации. Прямолинейное решение этой задачи заключается в создании самодельных сценариев, запускающих нужные контейнеры. Однако для реализации более гибкого решения, которое обеспечивало бы также готовность к эксплуатации, например, с нулевым временем простоя, целесообразно использовать фреймворки для управления контейнерами.

Фреймворки управления контейнерами, такие как *Kubernetes*, *DC/OS* и *Docker Compose*, позволяют не только запускать контейнеры, но и управлять ими, подключать и конфигурировать должным образом. Здесь действуют те же правила и принципы, что и для контейнерных технологий в целом: автоматизация, воспроизводимость и IaC. Инженеры-программисты определяют состояние, которое необходимо получить, в виде кода, и затем инструмент настройки по мере необходимости надежно устанавливает операционные среды.

Прежде чем перейти к описанию конкретных решений в области управления, рассмотрим подробнее саму концепцию.

Фреймворки управления позволяют объединять несколько контейнеров. Обычно это подразумевает поиск сервиса по логическому имени через DNS. Если используются несколько физических хостов, то фреймворк преобразует IP-адреса по этим узлам. В идеале приложение, работающее в контейнере, просто подключается к внешней системе, применяя логическое имя сервиса, которое преобразуется с помощью системы управления контейнера. Например, приложение для производства автомобилей, использующее базу данных транспортных средств `vehicle`, может соединяться с ней по имени хоста `vehicle-db`. Последнее затем преобразуется с помощью DNS в зависимости от среды, в которой работает приложение. Подключение посредством логических имен уменьшает объем конфигурационных данных в коде приложения, так как настроенное соединение

всегда будет одинаковым. В процессе управления будет просто подключен нужный экземпляр.

Это справедливо для всех предлагаемых систем. Приложения, базы данных и другие серверы абстрагируются до имен логических сервисов, по которым к ним обращаются и которые преобразуются во время выполнения.

Еще одна задача, которую решают фреймворки управления контейнерами, — это конфигурирование контейнеров в зависимости от среды. В целом рекомендуется уменьшить объем необходимой конфигурации в приложениях. Однако случается, что какая-то конфигурация необходима. Тогда можно воспользоваться фреймворком, чтобы создать конфигурацию контейнера путем динамического внедрения файлов или переменных среды в зависимости от обстоятельств.

Одним из самых больших преимуществ, предоставляемых некоторыми фреймворками управления, являются функции готовности к эксплуатации. При непрерывной разработке приложения постоянно создаются новые сборки проекта, что дает новые версии образов контейнеров. Действующие контейнеры необходимо заменять теми, которые запускаются из этих новых версий. Для того чтобы избежать простоев, фреймворк управления контейнерами меняет местами работающие контейнеры методом развертывания с нулевым простоем.

Точно так же фреймворк управления контейнерами позволяет повысить рабочую нагрузку, увеличив количество экземпляров контейнера. Прежде некоторые приложения выполнялись одновременно в нескольких экземплярах контейнеров. Если нужно было увеличить количество экземпляров, то требовалось больше серверов приложений. В эпоху контейнеров для достижения этой цели достаточно просто запустить больше контейнеров приложений без сохранения состояния. Программисты увеличивают количество реплик контейнеров в конфигурации, фреймворк управления реализует это изменение, запуская больше экземпляров контейнера.

Для того чтобы контейнеры работали в условиях промышленной эксплуатации, необходимо учесть некоторые тонкости управления ими. Как показывает опыт, некоторые компании склонны создавать собственные решения, а не использовать утвержденную технологию. Однако фреймворки управления контейнерами уже хорошо решают проблемы, и очень желательно по крайней мере рассмотреть вариант с их применением.

Реализация управления контейнерами

Итак, теперь мы знаем, какие проблемы решают фреймворки управления контейнерами. В этом разделе будут показаны основные концепции *Kubernetes* — решения, разработанного компанией Google для своих рабочих процессов. На момент написания этой книги Kubernetes резко набирал популярность и стал основой для других фреймворков управления, таких как *OpenShift* от RedHat. Я выбрал это решение не только из-за его популярности, но и потому, что считаю: оно отлично справляется с задачей управления. Однако в момент выбора главное — понимать не только технологию, но и причины и концепции ее применения.

Kubernetes запускает контейнеры Linux и управляет ими в кластере узлов. Главный узел Kubernetes управляет рабочими узлами, которые и выполняют фактическую работу, то есть в которых работают контейнеры. Инженеры-программисты управляют кластером с помощью API, предоставляемого главным узлом, с помощью графического веб-интерфейса или из командной строки.

Работающий кластер состоит из так называемых ресурсов определенного типа. Основными типами ресурсов Kubernetes являются *модули*, *развертывания* и *сервисы*. Модуль (pod) представляет собой минимальный блок рабочей нагрузки, в котором работают один или несколько контейнеров Linux. Таким образом, приложение работает в модуле.

Модули могут запускаться и управляться как автономные, отдельные ресурсы. Однако имеет смысл создавать не просто отдельные модули, а развертывание, которое инкапсулирует работающие модули и управляет ими. Развертывания предоставляют функциональные возможности, обеспечивающие готовность к эксплуатации, такие как укрупнение и разукрупнение модулей и развертывание обновлений. Они обеспечивают надежную работу указанных версий приложений.

Сервисы создаются в системе для подключения к работающим приложениям извне кластера или в других контейнерах. Они представляют собой описанную в предыдущем разделе логическую абстракцию, которая охватывает набор модулей. Все модули, в которых работает конкретное приложение, абстрагируются одним сервисом, который направляет трафик на активные модули. Маршрутизация сервисов к активным модулям в сочетании с развертываниями, управляющими последовательным обновлением версий, обеспечивает развертывание с нулевым временем простоя. Доступ к приложениям обеспечивают сервисы, которые направляют клиентов на соответствующие модули.

Все основные ресурсы уникальны в *пространстве имен* Kubernetes. Пространства имен инкапсулируют агрегаты ресурсов и могут применяться для моделирования различных сред. Например, сервисы, ссылающиеся на внешние системы, расположенные вне кластера, могут иметь разную конфигурацию в разных пространствах имен. Приложения, работающие с внешними системами, всегда используют одно и то же логическое имя сервиса, который направляет клиентов в разные конечные точки.

Kubernetes поддерживает определение ресурсов как IaC с использованием файлов JSON и YAML. Формат YAML представляет собой удобный для чтения формат сериализации данных, расширенный вариант JSON. В Kubernetes он стал фактическим стандартом.

В следующем фрагменте кода представлено определение службы приложения `hello-cloud`:

```
---
kind: Service
apiVersion: v1
metadata:
    name: hello-cloud
spec:
    selector:
```

```
        app: hello-cloud
    ports:
        - port: 8080
---
```

В этом примере описана служба, направляющая трафик на порт `8080` к модулям `hello-cloud`, которые определяются развертыванием.

В следующем коде описано развертывание приветствия `hello-cloud`:

```
---
kind: Deployment
apiVersion: apps/v1beta1
metadata:
    name: hello-cloud
spec:
    replicas: 1
    template:
        metadata:
            labels:
                app: hello-cloud
        spec:
            containers:
            - name: hello-cloud
                image: docker.example.com/hello-cloud:1
                imagePullPolicy: IfNotPresent
                livenessProbe:
                    httpGet:
                        path: /
                        port: 8080
                readinessProbe:
                    httpGet:
                        path: /hello-cloud/resources/hello
                        port: 8080
        restartPolicy: Always
---
```

В развертывании указан один модуль из заданного шаблона с образом, предоставленным Docker. Как только будет создано развертывание, Kubernetes проверяет соответствие спецификациям модуля, запустив контейнер из образа и протестировав его с помощью заданных зондов.

Образ контейнера `docker.example.com/hello-cloud:1` включает в себя корпоративное приложение, ранее собранное и развернутое через реестр Docker.

Все эти определения ресурсов применяются к кластеру Kubernetes либо с помощью графического веб-интерфейса, либо из командной строки.

После создания развертывания и службы приложение hello-cloud становится доступно изнутри кластера через сервис. Для доступа извне кластера необходимо определить маршрут, например используя вход `ingress`. Входные ресурсы направляют трафик на сервисы по определенным правилам. Далее приведен пример входных ресурсов, которые предоставляют доступ к сервису `hello-cloud`:

```
---
kind: Ingress
apiVersion: extensions/v1beta1
```

```
metadata:
    name: hello-cloud
spec:
    rules:
    - host: hello.example.com
        http:
            paths:
            - path: /
                backend:
                    serviceName: hello-cloud
                    servicePort: 8080
---
```

Эти ресурсы теперь описывают все приложение, которое развертывается в кластере Kubernetes, является доступным извне и абстрагируется как логический сервис внутри кластера. Если другим приложениям необходимо обмениваться данными с этим приложением, то они могут сделать это через разрешенные внутри DNS-домена имя `hello-cloud` Kubernetes и порт `8080`.

На рис. 5.3 показана примерная установка приложения hello-cloud с репликой из трех модулей, которые выполняются в кластере Kubernetes, состоящем из двух узлов.

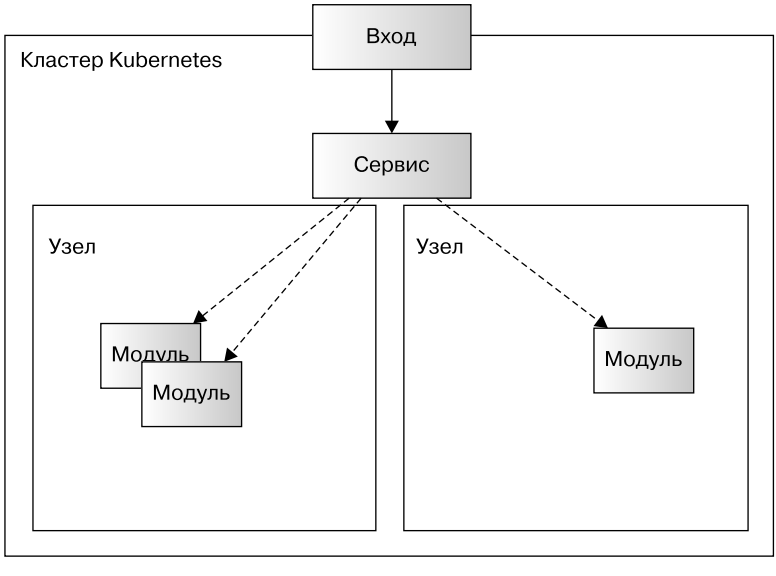

Рис. 5.3

Кроме поиска сервиса по логическому имени, некоторым приложениям по-прежнему требуется дополнительная настройка. Поэтому в Kubernetes, как и в других технологиях управления кластерами, есть возможность динамической вставки в контейнер файлов и переменных среды. Для этого применяется концепция *конфигурационных карт* (конфигурация на основе пар «ключ — значение»). Содержимое конфигурационных карт может быть доступно в виде файлов, ди-

намически устанавливаемых в контейнер. Далее приведен пример конфигурационной карты ConfigMap, определяющей содержимое файла свойств:

```
---
kind: ConfigMap
apiVersion: v1
metadata:
    name: hello-cloud-config
data:
    application.properties: |
        hello.greeting=Hello from Kubernetes
        hello.name=Java EE
---
```

Конфигурационные карты применяются для вставки содержимого в контейнеры в виде файлов. Ключи конфигурационной карты задействуются в качестве имен файлов, устанавливаемых в каталог, а значения являются содержимым этих файлов. В определениях модулей указывается, что конфигурационные карты смонтированы в виде томов. В следующем примере показано представленное ранее определение развертывания приложения hello-cloud с использованием конфигурационной карты hello-cloud-config в смонтированном томе:

```
---
kind: Deployment
apiVersion: apps/v1beta1
metadata:
    name: hello-cloud
spec:
    replicas: 1
    template:
        metadata:
            labels:
                app: hello-cloud
        spec:
            containers:
            - name: hello-cloud
                image: docker.example.com/hello-cloud:1
                imagePullPolicy: IfNotPresent
                volumeMounts:
                - name: config-volume
                    mountPath: /opt/config
                livenessProbe:
                    httpGet:
                        path: /
                        port: 8080
                readinessProbe:
                    httpGet:
                        path: /hello-cloud/resources/hello
                        port: 8080
            volumes:
            - name: config-volume
                configMap:
                    name: hello-cloud-config
            restartPolicy: Always
---
```

Развертывание определяет том, который ссылается на конфигурационную карту `hello-cloud-config`. Том монтируется в каталоге `/opt/config`, в результате чего все пары «ключ — значение» конфигурационной карты преобразуются в файлы и размещаются в этом каталоге. Так, согласно представленной ранее конфигурационной карте, будет создан файл `application.properties`, содержащий значения ключей `hello.greeting` и `hello.name`. Приложение ожидает, что во время выполнения файл будет находиться именно в этом месте.

Для разных сред могут применяться разные конфигурационные карты с разным содержимым в зависимости от желаемых значений конфигурации. Но есть и другие способы конфигурирования приложений, кроме использования динамических файлов. Также возможно вводить и переопределять конкретные переменные среды, как показано в следующем примере кода. Этот способ рекомендуется тогда, когда количество значений конфигурации ограниченно:

```
# как в предыдущем примере
# …
        image: docker.example.com/hello-cloud:1
        imagePullPolicy: IfNotPresent
        env:
        - name: GREETING_HELLO_NAME
            valueFrom:
                configMapRef:
                    name: hello-cloud-config
                    key: hello.name
            livenessProbe:
# …
```

Приложениям необходимо указать в конфигурации учетные данные, используемые, например, для авторизации во внешних системах или получения доступа к базам данных. В идеале эти учетные данные лучше хранить в другом месте, отдельно от некритических параметров конфигурации. Поэтому, кроме конфигурационных карт, в Kubernetes реализована концепция *секретов*. Как и конфигурационные карты, секреты представляют собой пары «ключ — значение», но недоступны для чтения человеком, так как зашифрованы по алгоритму Base64. Секреты и их содержимое обычно не сериализуются в виде инфраструктуры как кода, поскольку доступ к учетным данным должен быть ограничен.

Обычной практикой является предоставление доступа к учетным данным в контейнерах с помощью переменных среды. В следующем фрагменте кода показано, как включить в приложение hello-cloud значение, конфигурация которого хранится в секрете `hello-cloud-secret`:

```
# как в предыдущем примере
# …
        image: docker.example.com/hello-cloud:1
        imagePullPolicy: IfNotPresent
        env:
        - name: TOP_SECRET
            valueFrom:
                secretKeyRef:
                    name: hello-cloud-secret
```

```
                key: topsecret
          livenessProbe:
# …
```

Переменная окружения `TOP_SECRET` создается путем обращения к ключу `topsecret` в секрете `hello-cloud-secret`. Эта переменная среды доступна во время выполнения контейнера и может использоваться действующим процессом.

Некоторые приложения, упакованные в контейнеры, не могут выполняться отдельно, как приложения без сохранения состояния. Типичный пример — базы данных. Поскольку после завершения процессов соответствующие контейнеры отбрасываются, содержимое их файловой системы также исчезает. Однако такие сервисы, как базы данных, нуждаются в сохранении состояния. Для решения этой проблемы в Kubernetes используются *постоянные тома*. Как видно из названия, они доступны после окончания жизненного цикла модулей. Постоянные тома динамически делают доступными файлы и каталоги, которые применяются в модуле, и сохраняют их после завершения его работы.

Постоянные тома поддерживаются сетевыми или облачными хранилищами в зависимости от того, где установлен кластер. Они позволяют запускать сервисы хранения, такие как базы данных, в кластерах с управлением контейнерами. Однако, как правило, следует избегать постоянного хранения состояния в контейнерах.

Определения YAML IaC хранятся в системе контроля версий в репозитории приложения. В следующей главе будет показано, как в кластере Kubernetes применять содержимое файла как часть конвейера непрерывной поставки.

Java EE в управляемых контейнерах

Фреймворки управления настраивают и интегрируют корпоративные приложения в кластерных средах. Это экономит много сил при разработке приложения. Управление контейнерами также значительно упрощает конфигурацию приложения и способы его подключения к внешним сервисам — как именно, мы рассмотрим в данном разделе.

Подключение к внешним сервисам

Для интеграции внешних служб клиентским элементам управления требуется соединение посредством URL-адресов. Традиционно конфигурация URL-адресов описывалась в файлах, которые могли различаться в разных средах. В управляемой среде приложение может выполнять преобразование адресов внешних служб по их логическим именам через DNS. В следующем фрагменте кода показано подключение приложения cloud-processor:

```
@ApplicationScoped
public class HelloCloudProcessor {

    private Client client;
    private WebTarget target;
```

```
    @PostConstruct
    private void initClient() {
        client = ClientBuilder…
        target = client.target("http://cloud-
            processor:8080/processor/resources/hello");
    }

    public String processGreeting() {
        …
    }
}
```

Это справедливо и для других URL-адресов, например для определения источников данных. В конфигурации сервера приложений можно просто указать имя сервиса базы данных и использовать его для получения ссылки на соответствующий экземпляр во время выполнения приложения.

Конфигурирование управляемых приложений

Обнаружение сервисов по логическим именам позволяет значительно сократить объем конфигурации в приложении. Поскольку один и тот же образ контейнера используется во всех средах, потенциальные различия конфигурационных данных будут передаваться из среды управления. Как было показано в предыдущем примере, конфигурационные карты Kubernetes справляются с этой задачей. Приложение hello-cloud ожидает, что во время выполнения файл свойств будет находиться по адресу `/opt/config/application.properties` и именно туда будет обращаться код проекта. Далее показано, как интегрировать файл свойств в CDI-генератор:

```
public class HelloGreeter {

    @Inject
    @Config("hello.greeting")
    String greeting;

    @Inject
    @Config("hello.name")
    String greetingName;

    public String processGreeting() {
        return greeting + ", " + greetingName;
    }
}
```

Описание CDI-генератора аналогично приведенному ранее примеру с конфигурацией:

```
@ApplicationScoped
public class ConfigurationExposer {

    private final Properties properties = new Properties();

    @PostConstruct
```

```
    private void initProperties() {
        try (InputStream inputStream =
                new FileInputStream("/opt/config/application.properties")) {
            properties.load(inputStream);
        } catch (IOException e) {
            throw new IllegalStateException("Не удалось инициализировать
                конфигурацию", e);
        }
    }

    @Produces
    @Config("")
    public String exposeConfig(InjectionPoint injectionPoint) {
        Config config =
            injectionPoint.getAnnotated().getAnnotation(Config.class);
        if (config != null)
            return properties.getProperty(config.value());
        return null;
    }
}
```

Определение квалификатора `@Config` аналогично примеру, приведенному в главе 3. Приложение загружает содержимое файла свойств в карту свойств и генерирует параметры конфигурации с помощью CDI. Все управляемые компоненты могут внедрять эти параметры, получаемые из конфигурационной карты Kubernetes.

Для того чтобы реализовать секретные параметры конфигурации, в Kubernetes включена описанная ранее концепция секретов. Обычная практика заключается в том, чтобы обеспечить доступ к содержимому секретов в контейнерах через переменные среды.

Доступ к переменным среды в приложениях Java получают посредством метода `System.getenv()`. Эта функциональность используется как для секретов, так и для значений конфигурационной карты.

Продемонстрированные принципы и примеры позволяют развертывать корпоративные приложения в кластере, управляющем контейнерами, управлять ими и настраивать их. Для большинства сценариев этого достаточно.

Двенадцатифакторные приложения и Java EE

На момент написания этой книги *12-факторные* приложения стали способом разработки приложений типа *программное обеспечение как сервис (Software as a Service, SaaS)*. Концепция 12-факторного приложения определяет 12 принципов разработки программного обеспечения. Цель этих принципов — свести к минимуму время и усилия, затрачиваемые на создание программного обеспечения, избежать его эрозии, обеспечить возможность непрерывной поставки и применения на облачных платформах.

Другими словами, эти 12 факторов способствуют современному внедрению корпоративных приложений. Некоторые из них очевидны для большинства

инженеров, в то время как другие, по-видимому, противоречат общей практике создания корпоративных приложений.

Вот эти 12 факторов.

1. Одна кодовая база, отслеживаемая в системе контроля версий, и множество развертываний.
2. Явно объявляйте и изолируйте зависимости.
3. Храните конфигурацию в среде выполнения.
4. Рассматривайте вспомогательные сервисы как подключаемые ресурсы.
5. Строго разделяйте этапы сборки и запуска.
6. Выполняйте приложение как один или несколько процессов без сохранения состояния.
7. Экспортируйте сервисы через привязку портов.
8. Выполняйте масштабирование с помощью процессов.
9. Обеспечьте максимальную надежность, быстрый запуск и плавное отключение.
10. Сделайте разработку, установку и запуск в эксплуатацию максимально похожими.
11. Обрабатывайте журналы как потоки событий.
12. Запускайте задачи администрирования и управления как однократные процессы.

Далее мы рассмотрим причины внедрения всех этих принципов и их реализацию в Java EE.

Одна кодовая база в системе контроля версий и множество развертываний

Для инженеров-программистов этот принцип совершенно очевиден: программный код должен храниться в системе контроля версий в виде одного репозитория, даже если *развертываний несколько*. Развертывания — это экземпляры программного обеспечения, работающие в различных средах. Поэтому база кода приложения должна храниться в одном репозитории, а не представлять собой несколько версий кода для каждого приложения, и содержать все спецификации для потенциально разных сред выполнения.

Этот принцип позволяет повысить продуктивность труда программистов, поскольку вся информация находится в одном хранилище. Он не зависит от выбранной технологии и, следовательно, поддерживается приложениями Java EE.

Репозиторий должен содержать все исходные файлы, необходимые для сборки и запуска корпоративного приложения. Помимо источников Java и файлов конфигурации, сюда входит также инфраструктура как код.

Явное объявление и изоляция зависимостей

Программные зависимости и их версии, необходимые для запуска приложения, должны быть указаны явно. Это касается не только зависимостей, запрограммированных в приложении, таких как сторонние API, но и неявных зависимостей от среды выполнения Java или операционной системы. Когда требуемые версии указаны явно, возникает гораздо меньше проблем совместимости во время эксплуатации. Состав программных версий довольно хорошо тестируется в процессе разработки. Если при повторной сборке двоичных файлов версии зависимостей различаются, могут возникнуть проблемы. Поэтому рекомендуется явно указывать все версии программного обеспечения, чтобы уменьшить вероятность ошибок и обеспечить воспроизводимость.

Контейнерная технология упрощает реализацию этого принципа, явно описывая все этапы установки программного обеспечения. Версии используемых базовых образов должны быть объявлены явно, так что каждая сборка образа приводит к одному и тому же результату. Поэтому в Docker следует избегать тега `latest`, указывая вместо этого конкретные версии. Во всех установках программного обеспечения, описанных в Dockerfiles, также должны быть указаны явные версии. Повторные сборки Docker, с кэшем или без него, должны приводить к одному и тому же результату.

В приложениях Java зависимости определяются посредством систем сборки. В главе 1 уже говорилось о том, что нужно для создания воспроизводимых сборок с применением Maven и Gradle. В приложениях Java EE эти зависимости в идеале сводятся к API Java EE.

По возможности рекомендуется указывать версии зависимостей явно, а не просто использовать тег `latest`. Только программное обеспечение с явным указанием версий может быть надежно протестировано.

Изоляция зависимостей необходима для осуществления распределенной разработки командой программистов. Доступ к программным артефактам должен быть обеспечен посредством четко описанных процессов, например, в репозиториях артефактов. Зависимости, добавляемые во время сборки программного обеспечения, будь то среда выполнения Java, артефакты Java или компоненты операционной системы, должны распределяться из центрального хранилища. Такой подход можно реализовать с помощью репозиториев, таких как *Maven Central* и *DockerHub*, или корпоративных хранилищ.

Хранение конфигурации в среде

Конфигурационные данные приложения, зависящие от среды, такие как параметры баз данных, внешних систем или учетных записей, должны храниться в среде выполнения. Они не должны отражаться в исходном коде, а вместо этого динамически модифицироваться извне. Это означает, что конфигурация извлекается из файлов, переменных среды или других внешних источников.

Как уже было показано ранее, контейнерные технологии и структуры управления поддерживают эти концепции. Конфигурация для различных сред, таких как *среда тестирования*, *промежуточная среда* и *среда эксплуатации*, хранится в конфигурационных картах Kubernetes и динамически используется в модулях томов или в переменных окружения.

Согласно 12-факторной концепции приложение хранит конфигурацию в переменных среды. Переменные среды — это простой способ вставки определенных вариантов, поддерживаемых всеми видами технологий. Но если конфигурация приложения включает в себя множество индивидуальных значений, инженеры могут вместо этого использовать файлы конфигурации в томах контейнера.

Вспомогательные сервисы как подключаемые ресурсы

Базы данных и внешние системы, к которым обращается приложение, называются ресурсами. Для системы не должно иметь значения, является внешний сервис или база данных частью приложения или нет. *Ресурсы* должны подключаться к приложению методом слабого связывания. Должна быть возможность заменить внешние системы и базы данных новыми экземплярами, не затрагивая само приложение.

Приложения абстрагируют подключенную к ним внешнюю систему, прежде всего на уровне применяемой коммуникационной технологии. Так, связь через HTTP или JDBC абстрагирует реализации и позволяет заменять одни системы другими. Таким образом, приложения связаны только с их контрактом — протоколом связи и определенными схемами. Примерами поддержки этой концепции являются JPA, JAX-RS и JSON-B.

Фреймворки управления контейнерами продвинулись еще дальше, они абстрагируют сервисы, заменяя их логическими именами. Как было показано ранее, приложения могут использовать имена сервисов в качестве имен хостов, разрешаемых посредством DNS.

В целом, при разработке приложений следует свободно связывать системы, в идеале оставляя только зависимости от протоколов и схем. На уровне кода службы поддержки абстрагируются в собственные компоненты, такие как отдельные элементы управления с чистыми интерфейсами. Это сводит к минимуму изменения кода в случае, если изменятся связанные ресурсы.

Строгое разделение этапов сборки и запуска

Эта концепция рекомендует разделить процессы сборки, развертывания и запуска приложений. Она хорошо знакома разработчикам корпоративных Java-приложений. Существуют отдельные этапы сборки, развертывания и выполнения бинарных файлов приложений. Изменение программного обеспечения или его конфигурации происходит в исходном коде или на этапе развертывания соответственно, но не прямо в процессе эксплуатации. На этапе развертывания

двоичные файлы приложений объединяются с потенциальной конфигурацией. Хорошо описанные процессы управления изменениями и выпусками обеспечивают целостность корпоративного программного обеспечения.

Для подавляющего большинства программных проектов распространенной практикой является разделение этих этапов и управление ими на сервере непрерывной интеграции. Это необходимо для обеспечения надежности и воспроизводимости. Эти вопросы подробно описаны в главе 6.

Выполнение приложения как одного или нескольких процессов без сохранения состояния

В идеале приложения выполняются как процессы без сохранения состояния, так что каждый сценарий использования протекает отдельно, не затрагивая другие запущенные процессы. Потенциально состояние либо сохраняется в прикрепленном ресурсе, таком как база данных, либо отбрасывается. Таким образом, состояние сессии длительнее одного запроса нарушает этот принцип. Проблема традиционного состояния пользовательской сессии состоит в том, что оно хранится только в экземпляре локального приложения и недоступно из других экземпляров. Необходимость так называемых *залипающих сессий* в распределителях нагрузки является индикатором того, что данный продукт не является приложением без сохранения состояния.

Такой подход характерен для многих современных технологий. Примерами являются контейнеры в Docker и их файловая система с копированием при записи. После остановки контейнеры отбрасываются, их состояние также теряется. На аналогичном принципе основаны EJB-компоненты без сохранения состояния. Однако экземпляры сессионных компонентов без сохранения состояния помещаются в накопитель и используются повторно, поэтому необходимо обеспечить, чтобы ни одно состояние не сохранялось после вызовов бизнес-сценариев.

Корпоративные приложения должны перезапускаться с нуля, не изменяя свое поведение. Это также подразумевает, что приложения обмениваются состояниями только через четко описанные подключаемые ресурсы.

Экспорт сервисов через привязку портов

Веб-приложения традиционно развертываются в виде определенного стека программного обеспечения. Так, корпоративные приложения Java развертываются в корпоративном контейнере или контейнере сервлетов, тогда как приложения, написанные на серверных сценарных языках, таких как PHP, работают поверх веб-сервера. Поэтому приложения зависят от их непосредственной среды выполнения.

Двенадцатифакторная концепция рекомендует разрабатывать самодостаточные приложения, функциональность которых реализуется через сетевые порты. Поскольку корпоративные веб-приложения будут взаимодействовать через сеть, привязка сервисов к портам является способом слабого связывания.

Приложения Java EE, которые работают в контейнере, поддерживают этот принцип, экспортируя данные только через порт, используемый для связи с приложением. Контейнеры зависят только от ядра Linux, поэтому среда выполнения приложений прозрачна. Фреймворки управления контейнерами реализуют эту идею, связывая сервисы с контейнерами через логические имена и порты, как было показано в предыдущем примере. Java EE поддерживает применение контейнеров, а следовательно, и этот принцип.

Масштабирование с помощью процессов

Современные приложения и их среды должны обеспечивать масштабируемость в случае увеличения рабочей нагрузки. В идеале приложения должны масштабироваться не по вертикали, а по горизонтали. Разница заключается в том, что масштабирование по горизонтали означает добавление к приложению новых индивидуальных автономных узлов, тогда как при масштабировании по вертикали увеличивается количество ресурсов для отдельных узлов или процессов. Однако масштабировать по вертикали можно ограниченно, поскольку количество ресурсов на физических узлах нельзя увеличивать бесконечно.

В 12-факторных приложениях процедура добавления параллелизма в программное обеспечение описывается как добавление новых автономных процессов *без совместного доступа*. Рабочие нагрузки должны распределяться между несколькими физическими хостами, увеличивая количество процессов. Процессы представляют собой запросы или рабочие потоки, обрабатывающие рабочую нагрузку системы.

Эта концепция показывает необходимость реализации приложений без сохранения состояния в режиме «ничего общего». Контейнеры, запускающие корпоративные Java-приложения без сохранения состояния, позволяют масштабировать систему. Kubernetes обеспечивает управляемую масштабируемость при развертывании путем управления количеством реплик.

Однако узким местом корпоративных приложений обычно являются не экземпляры приложений, а центральные базы данных. Подробнее о масштабируемости в распределенных системах, а также о производительности в проектах Java EE в целом вы узнаете в главах 8 и 9.

Максимальная надежность, быстрый запуск и плавное отключение

В главе 4 уже говорилось о том, что переключение должно быть быстрым. Концепция 12-факторных приложений требует применения технологии, обеспечивающей скорость и гибкость. Для того чтобы программное обеспечение быстро масштабировалось, оно должно запускаться за считаные секунды, чтобы справляться с растущей нагрузкой.

При завершении работы приложение должно плавно заканчивать текущие запросы и правильно закрывать все открытые соединения и ресурсы. Особенно это важно для запросов и транзакций, которые выполнялись в момент сигнала выключения, — они должны быть надлежащим образом завершены, чтобы не прерывалось выполнение клиентских сценариев. В концепции процессов Unix при окончании работы отправляется сигнал `SIGTERM`. Контейнеры Linux останавливаются одинаково, что позволяет контейнерным процессам нормально завершить работу. При сборке образов контейнеров программистам следует обратить внимание на правильность обработки процессом сигналов Unix — это позволит плавно завершать работу сервера приложений при получении сигнала `SIGTERM`.

Java EE поддерживает как быстрый запуск, так и плавное выключение. Как было показано ранее, современные серверы приложений запускают и развертывают приложения за считаные секунды.

Поскольку серверы приложений управляют компонентами, ресурсами, накопителем и потоками, они заботятся и о том, чтобы правильно закрыть ресурсы при остановке JVM. Программистам не приходится самостоятельно заботиться об этом. Компоненты, управляющие специальными ресурсами или дескрипторами, которые необходимо закрыть, задействуют методы предуничтожения для правильного закрытия. В следующем примере показан клиентский элемент управления с использованием клиентского дескриптора JAX-RS, который закрывается при завершении работы сервера:

```
@ApplicationScoped
public class CoffeePurchaser {

    private Client client;

    …

    @PreDestroy
    public void closeClient() {
        client.close();
    }
}
```

Платформа гарантирует, что методы предуничтожения всех управляемых компонентов будут вызваны при выключении сервера приложений.

Максимально единообразная разработка, установка и запуск в эксплуатацию

Этот из 12 факторов направлен на минимизацию различий между средами.

Среды процесса разработки корпоративных приложений традиционно различаются между собой. Существует несколько вариантов среды разработки, среди которых локальные рабочие станции и выделенные серверные среды, — и есть, наконец, среда эксплуатации. Они различаются по времени развертывания

программных артефактов определенных версий и конфигурации в процессе разработки. Чем дольше одновременно используются разные версии в разных средах, тем значительнее будут различия.

Сказывается здесь и то, что работой заняты различные команды и сотрудники. Традиционно разработчики программного обеспечения поддерживают среду разработки, а операционная группа заботится о среде эксплуатации. Это может стать причиной недостаточно тесной коммуникации, а также несовпадения процессов и используемых технологий.

Самый большой риск представляет собой техническая разница между средами. Если в средах разработки и тестирования применяются одни инструменты, технологии, внешние сервисы и конфигурации, а в сфере эксплуатации — другие, то эти различия могут вызвать появление ошибок. Перед запуском в эксплуатацию программное обеспечение автоматически тестируется в этих средах. Каждое непротестированное отличие от условий эксплуатации может привести и рано или поздно приведет к ошибкам, которые можно было предотвратить. То же самое верно при замене инструментария, серверных сервисов и используемых стеков их упрощенными альтернативами в среде разработки и локальных средах.

Поэтому рекомендуется работать в максимально похожих средах. Особенно важно это для контейнерных технологий и фреймворков управления. Как мы уже видели, различия в конфигурации, сервисах и технологии сведены к минимуму или по крайней мере явно определены через среду. В идеале программные среды при разработке, тестировании, переносе и эксплуатации должны быть идентичными. Если это невозможно, то различия поддерживаются абстракциями сервисов, а также управляемыми средой конфигурациями.

Проблема различий во времени и обслуживающих командах решается благодаря непрерывной поставке не только с технической, но и с организационной точки зрения. Общее время производства должно быть как можно меньше, что позволяет быстро устанавливать новые функции и исправлять ошибки. Внедрение технологии непрерывной поставки естественным образом объединяет подразделения и обязанности. Согласно принципам DevOps все инженеры отвечают за все программное обеспечение. Это приводит к тому, что все команды тесно сотрудничают или объединяются в общую группу инженеров-программистов.

Журналы как потоки событий

Корпоративные приложения традиционно вносят журнальные записи в файлы журналов, хранящиеся на диске. Некоторые инженеры утверждают, что эта информация является одним из самых важных источников сведений о приложении. Проект программного обеспечения обычно включает в себя конфигурацию содержимого и формат этих журнальных файлов. Однако хранение данных журнала событий в файлах рассчитано главным образом на чтение и обычно предусматривает размещение каждого журнального события в одной строке.

Согласно концепции 12-факторных приложений журналирование должно рассматриваться как поток событий журнала, генерируемых приложением.

Однако приложения не должны сами решать вопросы маршрутизации и хранения файла журнала в определенном выходном формате. Вместо этого они передают записи журнала на стандартный выход процесса. Этот выход перехватывается и обрабатывается средой выполнения.

Для большинства корпоративных разработчиков со всеми их фреймворками журналирования, выходными форматами и инструментами такой подход необычен. Однако среды выполнения, в которых множество сервисов работает параллельно, должны в любом случае фиксировать и обрабатывать события журнала. Такие решения, как *Elasticsearch*, *Logstash* и *Kibana*, хорошо себя зарекомендовали при обработке журнальных событий, поступающих из нескольких источников, в том числе в сложных ситуациях. Сохранение событий журнала в файлах журналов не всегда поддерживает эти концепции.

Вывод событий журнала на стандартный выход приложения не только упрощает разработку, поскольку маршрутизация и хранение больше не входят в круг задач приложения. Также это уменьшает необходимость внешних зависимостей, таких как фреймворки регистрации. Данный подход поддерживается приложениями с нулевой зависимостью. Среда выполнения, такая как фреймворк управления контейнерами, берет на себя перехват и маршрутизацию потока событий. Подробнее о ведении журнала, его необходимости и недостатках читайте в главе 9.

Запуск задач администрирования и управления как однократных процессов

Согласно этому принципу административные и управленческие задачи должны реализовываться как отдельные краткосрочные процессы. Эта технология идеально поддерживает выполнение команд в оболочке, которая работает в среде выполнения.

Хотя контейнеры инкапсулируют Unix-процессы, они предоставляют дополнительные функции для выполнения отдельных команд или открытия в контейнере удаленной оболочки. Благодаря этому инженеры могут реализовывать сценарии управления и администрирования, предоставляемые сервером приложений Java EE. Тем не менее в приложениях Java EE количество необходимых задач администрирования и управления ограничено. Контейнер запускает процесс сервера приложений, который автоматически развертывает приложение, и в дальнейшем управлять жизненным циклом приложения не требуется.

Большинство задач администрирования выполняется в процессе отладки и при устранении неполадок. Поэтому контейнеры и фреймворки управления контейнерами позволяют открывать удаленные оболочки с доступом в контейнеры или выполнять одноразовые команды. Кроме этого, в главе 9 будет показано, что необходимо для сбора дополнительной информации о корпоративных приложениях.

Описанные 12 факторов позволяют разрабатывать масштабируемые корпоративные приложения без сохранения состояния, обеспечивающие непрерывную поставку, работающие на современных корпоративных платформах, дающие

возможность оптимизировать время и усилия, затраченные на разработку, и избежать эрозии программного обеспечения. Двенадцатифакторные приложения имеют чистый контракт с базовой средой и в идеале декларативные определения инфраструктуры.

Облака, облачные приложения и их преимущества

На момент написания этой книги облачные платформы вызывали большой интерес. Мы наблюдаем, как крупные компании перемещают свою ИТ-инфраструктуру в облачные среды. Какие же преимущества предлагает облако?

Прежде всего следует знать, что современные среды не обязательно должны работать на облачной платформе. Все преимущества контейнерной технологии и фреймворка управления контейнерами доступны и во внутренней инфраструктуре предприятия. В стандартных установках таких платформ, как Kubernetes или OpenShift, обеспечиваются те же преимущества для команд разработчиков программного обеспечения. В сущности, одним из самых больших преимуществ контейнерной среды выполнения является абстрагирование среды, в которой работают контейнеры. Чем же тогда облачные платформы интересны для компаний?

Как отмечалось в начале книги, сегодня мир программного обеспечения меняется быстрее, чем когда-либо. Ключом к тому, чтобы компании не отставали от тенденций в своем бизнесе, являются скорость и гибкость, то есть быстрота изменений. Время выхода на рынок новых продуктов и функций должно быть как можно короче. Поэтапное продвижение, адаптация к потребностям клиентов и постоянное совершенствование программного обеспечения отвечают этому требованию. Для реализации этой цели ИТ-инфраструктура, как и остальные составляющие программного обеспечения, должна быть быстрой и гибкой. Установка новых сред должна быть автоматизированным, надежным и воспроизводимым процессом. Принципы непрерывной поставки программного обеспечения относятся и к серверным средам. Именно эту возможность предоставляют облачные платформы.

Если компания желает обеспечить гибкость и адаптироваться к требованиям клиентов, то ей следует задать себе вопрос: *сколько времени нужно, чтобы предоставить им новую среду выполнения*? Это необходимое условие быстрой адаптации. Предоставление совершенно новой среды выполнения не должно быть чрезмерно сложным процессом, должно занимать несколько минут и в идеале протекать без вмешательства человека. Как уже говорилось, такую концепцию вполне возможно реализовать локально. Однако облачные среды предлагают эти преимущества по умолчанию с достаточными и масштабируемыми ресурсами. *Инфраструктура как сервис (Infrastructure as a Service, IaaS)* или *платформа как сервис (Platform as a Service, PaaS)* освобождают в компаниях много рабочих рук, что позволяет разработчикам сосредоточиться на создании собственных продуктов.

Несмотря на это, крупные компании часто скептически относятся к облачным сервисам, особенно опасаясь за сохранность данных. Интересно, что, как показывает опыт создания проектов, облачные платформы, на которых работают сложные корпоративные продукты, обеспечивают более безопасное окружение, чем большинство обычных решений. Поставщики облачных платформ тратят много времени и прикладывают большие усилия для построения правильных решений. Особенно значительным потенциалом обладают сочетания облачных платформ с решениями по управлению, такими как Docker Compose, Kubernetes и OpenShift.

Интересно, что для компаний одним из главных аргументов в пользу переноса ИТ в облако являются экономические причины. Как показывает опыт, многие компании хотят сэкономить, используя облачные платформы. В сущности, принимая во внимание весь процесс миграции и преобразования среды, команд, технологий и прежде всего всех ноу-хау, оставаться на старой платформе, как правило, дешевле. Однако основными преимуществами облачных предложений являются гибкость и способность быстро меняться. Если ИТ-компания создала хорошо управляемую среду, в том числе автоматизированные, надежные и воспроизводимые процессы, целесообразно сохранить ее и продолжать совершенствовать. Другими словами, современная среда — это не столько облачные платформы, сколько процессы, командный менталитет и разумное применение технологий.

Облачные приложения. Кроме интереса к облачным технологиям, существует также интерес к *облачным приложениям* — приложениям, которые не только соответствуют концепции 12 факторов, но и прочно связаны с облачными платформами. Облачные и 12-факторные приложения не являются синонимами, скорее, облачные приложения, кроме прочего, являются также 12-факторными.

Облачные приложения предназначены для работы в облачных средах PaaS со всеми их преимуществами и задачами, включая контейнерные технологии и гибкую масштабируемость. Они построены как современные масштабируемые и устойчивые приложения без сохранения состояния, управляемые современными средами управления. Несмотря на то что эти приложения называются облачными, они не всегда построены как проекты, развертываемые с нуля и поддерживающие технологию облачных вычислений по умолчанию.

Важными показателями облачных приложений, помимо 12 факторов, являются мониторинг и исправность приложений, которые в целом можно назвать телеметрией. Телеметрия для корпоративных приложений включает в себя быстрое реагирование, мониторинг, специфическую информацию о предметной области, проверку работоспособности и отладку. Как мы уже знаем, управление контейнерами решает как минимум две последние задачи — проверку работоспособности и отладку. Во время работы приложений проверяется, действуют ли они и правильно ли работают. Отладить и устранить неполадки можно путем оценки потоков событий журнала, подключения к работающим контейнерам или протекающим процессам.

Мониторинг приложений должен выполнять работающий контейнер. Это требует немного больше усилий от разработчиков программного обеспечения. Прежде всего бизнес-эксперты должны определить показатели, характерные

для данной предметной области. Именно эти показатели, важные для бизнес-подразделений, будет отображать приложение. Также для работающего приложения собирают технические показатели. Подробнее о мониторинге в современных средах читайте в главе 9.

В рассмотренные 12 факторов не входят API и безопасность, которые также имеют большое значение. SaaS-приложения взаимодействуют через открытые API, которые должны быть известны другим группам программистов. Характер и структура веб-сервисов должны быть документированы и согласованы в процессе разработки. Это особенно важно, если API HTTP не реализует Hypermedia. Приложения должны знать характер и структуру передаваемой информации, в идеале — как можно раньше в процессе разработки. Сюда также входят проверка подлинности и авторизация. Разработчики приложений должны знать о механизмах безопасности, которые им необходимо использовать, прежде чем передавать данные другим службам. В целом нежелательно озаботиться вопросами безопасности лишь после разработки. Подробнее об облачных средах и интеграции в них приложений Java EE читайте в главе 10.

Для того чтобы построить обобщающую технологию для облачных платформ, несколько поставщиков программного обеспечения сформировали *Cloud Native Computing Foundation* — основы вычислительных технологий для облачных приложений. Это часть Linux Foundation, представляющая собой основу для разработки облачных приложений с открытым исходным кодом. Сюда входят технологии, которые организуют, управляют, контролируют, отслеживают и другими способами поддерживают контейнерные *микросервисы*, работающие в современных средах. На момент написания этой книги примерами технологических проектов, входящих в Cloud Native Computing Foundation, были Kubernetes, Prometheus, OpenTracing и containerd.

Резюме

Оперативные задачи должны быть автоматизированы. Настройка среды выполнения приложений, включая инсталляции, параметры сети и конфигурацию, всегда должна обеспечивать один и тот же результат. Контейнерные технологии, а также инфраструктура как код поддерживают этот принцип, определяя, автоматизируя и распространяя инсталляции и конфигурацию программного обеспечения. Они обеспечивают быстрый и воспроизводимый способ восстановления программного обеспечения и систем.

Инфраструктура как код предоставляет описания, определяющие требуемую инфраструктуру и все зависимости как часть кода приложения, хранящуюся в системе контроля версий. Этот принцип соответствует идеям движения DevOps. Обязанность описать не только приложение, но и его среду выполнения вместе со всеми требованиями, объединяет разные команды компании. Все инженеры должны отвечать за качество программного обеспечения, предназначенного для реализации бизнес-логики.

Контейнерные технологии, такие как Docker, предоставляют функционал для единообразных операций сборки, управления и поставки контейнеров. Многоуровневая файловая система Docker с копированием при записи позволяет минимизировать время сборки и публикации за счет только повторного выполнения только тех этапов, которые были изменены. Приложения с нулевой зависимостью Java EE поощряют использование контейнерной технологии, отделяя логику приложения от ее реализации, так что изменяющийся уровень содержит только бизнес-код.

Фреймворки управления контейнерами, такие как Kubernetes, управляют контейнерами в течение всего жизненного цикла, а также параметрами сети и внешней конфигурацией. Они несут ответственность за поисковые сервисы, обеспечивают готовность к эксплуатации, к которой относятся развертывание с нулевым временем простоя, а также увеличение и уменьшение количества экземпляров приложения. Управление контейнерами поддерживает определения инфраструктуры как кода, в которых содержится конфигурация всей среды выполнения, необходимая для приложения.

Концепция 12-факторных и облачных приложений направлена на разработку современных корпоративных приложений с минимальными затратами времени и усилий, предотвращение эрозии программного обеспечения, а также поддержку непрерывной поставки и облачных платформ. Двенадцатифакторная концепция касается программных зависимостей, конфигурации, зависимых сервисов, среды выполнения, журналирования и администрирования. Аналогично концепция облачных приложений направлена на создание корпоративного программного обеспечения, хорошо работающего на облачных платформах, с поддержкой мониторинга, принципов устойчивости, исправности и безопасности. Поскольку эти подходы не связаны с конкретной технологией, они реализуются в том числе и с использованием Java EE. Мы изучили причины, по которым следует придерживаться этих принципов.

В следующей главе будет показано, как строить эффективные рабочие процессы разработки приложений, основанные на контейнерных технологиях.

6 Рабочие процессы создания приложений

Читая предыдущую главу, вы наверняка осознали всю важность быстрого изменения для компаний по разработке ПО. Это влияет на инфраструктуру и среду выполнения, а также на совместную работу команд инженеров. Причины создания современных сред — стремление добиться масштабируемости и гибкости, минимизировать время и усилия.

Но еще более важными, чем собственно инфраструктура, являются процессы разработки. Весь процесс от создания исходного кода до выпуска работающего приложения должен быть описан разумным и продуктивным способом. И здесь под быстрым изменением в быстро меняющемся мире подразумевается, что эти процессы выполняются автоматически и надежно, по возможности с минимальным вмешательством человека.

В этой главе мы рассмотрим следующие темы.

- Причины и необходимость непрерывной поставки.
- Что представляют собой производственные конвейеры.
- Как автоматизировать все операции.
- Как гарантировать и улучшить качество программного обеспечения.
- Необходимая культура и модели работы в команде.

Цели и обоснование построения продуктивных рабочих процессов

Быстрое изменение с точки зрения процессов разработки означает обеспечение оперативной обратной связи за счет коротких производственных циклов. Для того чтобы повысить производительность, программисты, разрабатывающие поведение приложения, должны своевременно проверять реализованные функции

и исправлять ошибки. В том числе речь идет и о времени, затрачиваемом на сборку, тестирование и развертывание программного обеспечения.

Ключевым условием построения продуктивных рабочих процессов является автоматизация. Инженеры-программисты должны тратить как можно больше времени на проектирование, реализацию и обсуждение бизнес-логики и как можно меньше — на решение сквозных и повторяющихся задач. Компьютеры предназначены для быстрого и надежного решения детерминированных и простых задач. Но ни одна машина не сравнится с человеком в творческом мышлении, умении генерировать идеи во время мозгового штурма. Поэтому простые процессы, требующие принятия большого количества решений, должно выполнять программное обеспечение.

Таким программным обеспечением являются прежде всего системы сборки. Они позволяют автоматизировать компиляцию, разрешение зависимостей и упаковку программных проектов. Еще шире эта концепция используется в серверах непрерывной интеграции. Они управляют всем процессом разработки, от сборки артефактов до автоматического тестирования и развертывания. Серверы непрерывной интеграции являются истиной в последней инстанции при поставке программного обеспечения. Они постоянно интегрируют работу всех программистов в едином центре, гарантируя, что проект в любой момент готов к поставке.

Непрерывная поставка — это продолжение концепции непрерывной интеграции, включающая в себя автоматическую поставку собранного ПО в заданные среды выполнения после каждой сборки. Поскольку изменения программного обеспечения следует должным образом проверить, прежде чем они поступят в эксплуатацию, сначала приложения развертываются в тестовых и промежуточных средах. Все действия по развертыванию должны гарантировать, что среда подготовлена, настроена и развернута так, как нужно. Автоматизированные и ручные сквозные тесты гарантируют, что программное обеспечение работает так, как ожидалось. Последующее развертывание в среде эксплуатации выполняется в *полуавтоматическом режиме* посредством ручного запуска автоматического развертывания.

Разница между непрерывной поставкой и непрерывным развертыванием заключается в том, что последнее предусматривает автоматическое развертывание каждой подтвержденной версии программного обеспечения в среде эксплуатации — разумеется, при условии, что все требования к качеству удовлетворены.

Все эти принципы сводят к минимуму необходимость вмешательства программистов, сокращают производственный цикл и повышают производительность.

В идеале непрерывная поставка поддерживает не только развертывание, но и надежные откаты. Существуют причины, по которым версии программного обеспечения иногда приходится откатывать, несмотря на то что они были заранее

проверены. В такой ситуации можно либо продвинуться вперед, например перейдя на новую версию, отменяющую недавние изменения, либо возвратиться к последнему рабочему состоянию.

Как отмечалось ранее, сборка программного обеспечения должна выполняться надежным способом. Все версии используемой технологии, такие как зависимости сборки и серверы приложений, должны указываться явно. Повторная сборка приложений и контейнеров всегда должна давать одинаковые результаты. Этапы процессов разработки также должны приводить к одному и тому же результату. Крайне важно, чтобы для эксплуатации развертывался именно тот артефакт приложения, который был перед этим проверен в тестовых средах. Далее в этой главе мы рассмотрим, как получать воспроизводимые, повторяемые и независимые сборки.

Для обеспечения надежности желательно использовать автоматизированные процессы. В частности, развертывание с помощью программного обеспечения, без вмешательства человека гораздо менее подвержено ошибкам. Все необходимые этапы процесса четко определены и неявно проверяются при каждом выполнении. Таким образом, автоматизированные процессы в конечном счете гораздо надежнее, чем выполняемые вручную.

Проверка и тестирование — важные условия непрерывной поставки. Как показывает опыт, подавляющее большинство программных тестов могут быть автоматическими. В следующей главе мы подробно рассмотрим эту тему. Качество программного проекта обеспечивается не только тестированием, но и постоянным контролем.

Рабочие процессы непрерывной поставки включают в себя все этапы, необходимые для эффективной и автоматизированной сборки, тестирования, поставки и развертывания программного обеспечения. Из этой главы вы узнаете, как строить эффективные рабочие процессы.

Реализация процессов разработки

Процессы непрерывной поставки состоят из нескольких этапов, одни выполняются последовательно, а другие — параллельно. Все этапы являются частью единого процесса сборки. Она запускается обычно после фиксации изменений или, точнее, после записи изменений кода в систему контроля версий.

Далее будут рассмотрены составляющие процесса непрерывной поставки. Это общие этапы, не зависящие от применяемой технологии.

На рис. 6.1 упрощенно показаны основные элементы конвейера непрерывной поставки. Этапы реализуются на сервере непрерывной интеграции с использованием внешних репозиториев, таких как система контроля версий, артефакт и контейнерные репозитории.

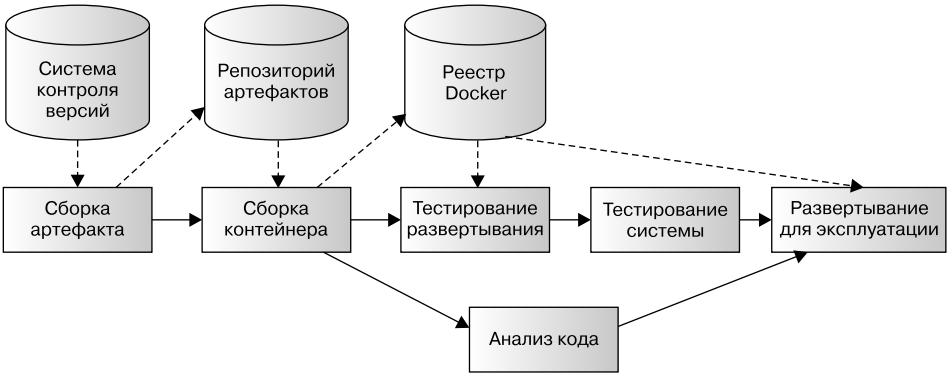

Рис. 6.1

Всё под контролем версий

Программисты подтверждают: исходный код должен храниться в системе контроля версий. Сейчас большой популярностью пользуются такие современные инструменты, как распределенные системы контроля версий типа Git. Однако, как уже отмечалось, помимо исходного кода приложения, есть еще объекты, изменение которых надо отслеживать.

Причина использования инфраструктуры как кода заключается в том, чтобы сохранить все артефакты, необходимые для поставки приложения, в едином центре. Все изменения, внесенные в приложение, конфигурацию или среду, представляются в виде кода и проверяются в репозитории. Инфраструктура как код позволяет повысить воспроизводимость и автоматизацию. При развитии этого подхода конвейеры непрерывной поставки также будут определены как код. Сам подход будет рассмотрен в разделе «Конвейер как код» на примере широко используемого сервера Jenkins.

Как говорилось в предыдущей главе, главным принципом 12-факторных приложений является хранение всех файлов и артефактов, необходимых для сборки и запуска приложения, в одном хранилище.

Первым шагом конвейера непрерывной поставки является проверка определенной фиксации изменений в системе контроля версий. Команды, использующие распределенные системы контроля версий, должны синхронизировать желаемое состояние с центральным хранилищем. Сервер непрерывной интеграции принимает состояние данной фиксации изменений, затем начинает процесс сборки.

Причиной использования конкретной фиксации изменений, а не просто последнего состояния, является гарантия воспроизводимости. Если сборка основана на конкретной фиксации изменений, то повторная сборка той же версии всегда будет приводить к одному и тому же результату. Это возможно только в том

случае, если сборка основана на определенной фиксации состояния в системе контроля версий. Операция записи в систему контроля версий обычно запускает сборку на основании соответствующей версии фиксации.

При чтении репозитория предоставляются все источники и файлы. Следующим шагом является сборка артефактов программного обеспечения.

Сборка двоичных файлов

Как говорилось в главе 1, к двоичным файлам относятся все исполняемые артефакты, которые запускают корпоративное приложение. Репозиторий проекта содержит только исходный код, а также файлы и артефакты, необходимые для инфраструктуры. Бинарные файлы собираются на сервере непрерывной интеграции.

Данный этап конвейера отвечает за надежную сборку этих двоичных файлов и обеспечение их доступности.

Артефакты Java

В Java EE основной двоичный файл — это упакованное корпоративное приложение в виде архива. В соответствии с принципом создания приложений с нулевой зависимостью проект собирается и упаковывается в «тонкий» WAR-файл, содержащий только бизнес-логику приложения. Операция сборки включает в себя разрешение требуемых зависимостей, компиляцию исходных кодов Java и упаковку двоичных классов и других файлов в архив. WAR-файлы являются первым создаваемым артефактом в конвейере сборки.

Артефакты приложения собирают с использованием таких систем сборки, как Maven или Gradle, которые устанавливаются на сервере непрерывной интеграции. Обычно сборка проекта включает в себя базовые тесты на уровне кода. Подобные тесты, которым не требуется использование среды выполнения контейнера, позволяют проверить поведение классов и компонентов на ранних этапах конвейера. Принцип непрерывной поставки подразумевает, что в случае возникновения ошибок нужно как можно раньше отменить и прервать сборку, чтобы максимально сократить производственный цикл.

При необходимости системы сборки могут публиковать артефакты в репозитории артефактов. Репозитории артефактов, такие как *Sonatype Nexus* или *JFrog Artifactory*, позволяют хранить собранные версии артефакта для последующего использования. Но если приложение поставляется в контейнерах Linux, то артефакт не обязательно развертывать в репозитории.

Как было показано в главе 2, проект Java собирается в Maven с помощью команды `mvn package`. На этапе упаковки собираются все исходные коды продукта Java, компилируются и запускаются исходные коды тестов и приложение упаковывается в данном случае в WAR-файл. Сервер непрерывной интеграции выполняет аналогичную команду сборки и собирает артефакт в локальном каталоге рабочей области. Этот артефакт можно развернуть в репозитории артефактов, например, с помощью команды `mvn deploy`, чтобы использовать его на следующих этапах. Или же его можно взять непосредственно из каталога рабочей области.

Версии артефактов

Как уже отмечалось, системы сборки должны обеспечивать надежность при создании артефактов. Для этого каждый артефакт Java должен собираться и архивироваться под своей версией, по которой его позже можно идентифицировать. При тестировании программного обеспечения проверяются конкретные версии корпоративных приложений. На последующих этапах сборки эти версии нужно указывать при развертывании. Необходима возможность идентифицировать различные версии артефакта и ссылаться на них. Это касается всех двоичных файлов.

Один из принципов 12-факторных приложений требует явного объявления зависимостей, в том числе используемых версий. Как уже отмечалось, это справедливо и для сборки контейнеров. Указанные базовые образы Docker, а также установленное программное обеспечение должны явно и однозначно идентифицироваться по их версиям.

Однако довольно часто указывают сборки Java как версии *снимков состояния*, например `0.1-SNAPSHOT`. Снимок состояния, в отличие от рабочей версии, представляет собой состояние программного обеспечения в процессе разработки. При разрешении зависимостей, если существует несколько версий снимка состояния, всегда делается попытка включить последний из них, как в Docker, по метке `latest`. Рабочий процесс на основе снимков состояния заключается в том, чтобы выпустить версию снимка состояния как версию с уникальным номером, как только ее уровень развития будет достаточным.

Однако создание версий из снимков состояния противоречит идее непрерывной поставки. В конвейерах непрерывной поставки каждая фиксация состояния является потенциальным кандидатом на развертывание в среде эксплуатации. Версии снимков состояния, разумеется, не предназначены для такого развертывания. Это означает, что в ходе рабочего процесса надо будет заменять снимок состояния на рабочую версию, после того как версия программного обеспечения пройдет необходимые проверки. Однако предполагается, что после сборки Java-артефакты не должны изменяться. Если артефакт прошел проверку, то именно он и должен использоваться для развертывания. Поэтому версии снимков состояния не подходят для конвейеров непрерывной поставки.

В соответствии с популярным принципом *семантического управления версиями* разработчикам приложений необходимо позаботиться об обратной совместимости версий. При семантическом управлении версиями программному обеспечению присваиваются такие номера версий, как 1.1.0, 1.0.0-beta или 1.0.1+b102. Версиям, имеющим право на непрерывную поставку, удобно присваивать номера с уникальными метаданными сборки, например 1.0.1+b102, где 1 — *основной* номер версии, 0 — *вспомогательный*, следующая 1 — номер версии *исправления* и 102 — номер *сборки*. То, что стоит после знака «плюс», — это необязательные метаданные сборки. Даже если семантическая версия не менялась на протяжении нескольких сборок, создаваемые артефакты по-прежнему можно идентифицировать. Артефакты можно опубликовать в репозитории артефактов и впоследствии получить оттуда по номерам версий.

Такой принцип управления версиями рассчитан на проекты корпоративных приложений, а не на продукты. Для продуктов, имеющих несколько поставляемых и поддерживаемых версий одновременно, требуются более сложные процессы управления версиями.

На момент написания этой книги еще не было фактического стандарта для управления версиями. Одни компании придерживаются принципа семантического управления версиями, тогда как другие используют только номера сборки сервера непрерывной интеграции или хешированные номера зафиксированных состояний. Все эти варианты можно использовать, если не перестраивать образ контейнера или не распространять приложение дважды по одной и той же метке. Каждая сборка должна давать отдельную версию образа контейнера.

Сборка контейнеров

Образы контейнеров — это тоже двоичные файлы, поскольку они содержат работающее приложение, в том числе среду выполнения и двоичные файлы операционной системы. Для того чтобы собирать образы контейнеров, необходимы базовые образы и все артефакты, которые добавляются во время сборки. Если их еще не существует в среде сборки, то базовые образы извлекаются неявно.

Для каждого этапа сборки, описанного в файле Docker, уровень образа добавляется поверх предыдущего уровня. Последним — по порядку, но не по значению — в сборку образа контейнера добавляется приложение, которое было собрано ранее. Как уже было показано, контейнеры приложений Java EE состоят из установленного и сконфигурированного сервера приложений, который автоматически развертывает веб-архив в среде выполнения.

Сервер непрерывной интеграции управляет этой сборкой образа как частью конвейера. Одним из решений является установка среды выполнения Docker аналогично системе сборки Maven. Затем реализуется этап конвейера, на котором в каталоге рабочей области используется команда сборки образа, примерно такая: `docker build -t docker.example.com/hello-cloud:1`. Например, при сборке образа Docker может взять WAR-файл в каталоге `target` Maven и добавить его в контейнер.

Собранному образу присваиваются имя и уникальная метка, содержащая номер сборки или другую уникальную информацию. Имена образов Docker включают в себя реестр, в который будут добавлены. Идентификатор образа, такой как `docker.example.com/hello-cloud:1`, будет неявно передаваться с узла `docker.example.com` и на него. Конвейер передает образ в реестр Docker, как правило, определяемый конкретной компанией.

Образы Docker могут быть повторно снабжены метками как часть конвейера, если это предусматривается рабочим процессом компании. Например, специальная метка `latest` может указывать на *последнюю собранную* версию. Для этого можно явно переименовать образ, чтобы два идентификатора указывали на один и тот же образ. В отличие от Java-архивов образы Docker можно повторно снабжать метками, не меняя их содержимого. Вторую метку также нужно добавить

в репозиторий. Однако, как мы увидим в дальнейшем в этой главе, не обязательно ссылаться на *последние версии образов*, такие как метка Docker `latest`. В сущности, как и при управлении версиями со снимками состояния, рекомендуется избегать *последних* версий. Лучше указывать явно все версии артефакта — так будет меньше вероятность ошибок.

Некоторые инженеры утверждают, что запуск Docker внутри сервера непрерывной интеграции — не лучшая идея, если сам CI-сервер работает как контейнер Docker. При сборке образа Docker запускаются временные контейнеры. Конечно, можно либо запускать контейнеры в контейнере, либо подключать среду выполнения к другому узлу Docker, не открывая всю платформу и не рискуя получить проблемы с безопасностью. Однако некоторые компании предпочитают собирать образы за пределами сервера непрерывной интеграции. Например, OpenShift — система PaaS, построенная на основе Kubernetes, обеспечивает функциональность, которая включает в себя сервер непрерывной интеграции, а также сборку образов. Итак, можно управлять сборкой образов с сервера непрерывной интеграции, а сама сборка будет осуществляться на платформе OpenShift, что является альтернативой сборке образов контейнеров непосредственно на CI-сервере.

Гарантия качества

Сборка артефактов Java сама по себе уже дает некую базовую гарантию качества. При сборке выполняются тесты на уровне кода, например модульные тесты. Правильно построенный конвейер включает в себя несколько тестовых показателей и сценариев тестирования, каждый из которых имеет и сильные и слабые стороны. Включенные в процесс сборки модульные тесты работают на уровне кода и могут быть проведены без какой-либо дополнительной рабочей среды. Их цели — проверить поведение отдельных классов и компонентов и предоставить быструю обратную связь в случае сбоев тестирования. В следующей главе мы увидим, что модульные тесты должны выполняться самостоятельно и быстро.

Результаты тестов обычно записываются с сервера непрерывной интеграции с целью обеспечения доступности и мониторинга. Сделать результат выполнения этапов конвейера видимым — важная составляющая непрерывной поставки. CI-сервер может отслеживать количество пройденных модульных тестов и показывать, что происходит с течением времени.

Существуют расширения системы сборки, которые отслеживают покрытие кода тестами. Покрытие показывает, какие части базы кода были выполнены при прохождении тестов. Вообще говоря, чем больше покрытие кода, тем лучше. Однако высокий процент покрытия кода сам по себе ничего не говорит о качестве тестов и покрытии тестовых утверждений. Результаты тестирования в сочетании с тестовым покрытием являются лишь одним из нескольких показателей качества.

Исходный код сам по себе может дать много информации о качестве программного обеспечения. Так называемый *статический анализ кода* подразумевает

определенную проверку качества статических файлов с исходным кодом проекта без их выполнения. При таком анализе собирают информацию об операторах кода, размерах классов и методов, зависимостях между классами и пакетами, сложности методов. Анализ статического кода позволяет найти возможные ошибки в исходном коде, например выявить ресурсы, не закрытые должным образом.

Одним из самых известных инструментов проверки качества кода является *SonarQube*. Он предоставляет информацию о качестве программных проектов, сопоставляя результаты различных методов анализа, таких как статический анализ кода и покрытие тестами. Объединенная информация позволяет сформировать показатели качества, полезные для инженеров-программистов и разработчиков. Например, какие методы сложны, но в то же время достаточно протестированы? Какие компоненты и классы самые большие по размеру и сложности и поэтому являются первыми кандидатами на реструктуризацию? Какие пакеты имеют циклические зависимости и, вероятно, содержат компоненты, которые следует объединить? Как тестовое покрытие меняется со временем? Сколько предупреждений и сообщений об ошибках выдал анализатор кода и как это количество изменяется со временем?

Рекомендуется придерживаться некоторых базовых принципов, касающихся статического анализа кода. Отдельные показатели просто позволяют составить приблизительное представление о качестве программного обеспечения. Типичный пример — тестовое покрытие. Высокое тестовое покрытие проекта не обязательно означает, что данное ПО хорошо протестировано: операторы контроля могут оказаться нецелесообразными или их может быть недостаточно. Однако представление о качестве дает анализ организации тестового покрытия — например, добавляются ли тесты программного обеспечения для новых и существующих функций, а также для исправления ошибок.

Существуют также показатели, которых следует строго придерживаться. К ним относятся, в частности, предупреждения и сообщения об ошибках анализатора кода. Они многое говорят инженерам о стиле и нарушениях качества кода и указывают на проблемы, которые необходимо устранить.

Прежде всего, не должно быть таких вещей, как предупреждения компиляции или анализа. Одно из двух: либо сборка успешно проходит все проверки качества и *получает зеленый свет*, либо качество недостаточно для развертывания — *красный свет*. Никаких компромиссов, третьего не дано. Рразработчики программного обеспечения должны ясно понимать, какие проблемы могут и должны быть решены, а какие — нет. Поэтому предупреждения, указывающие на незначительные проблемы в проекте, следует рассматривать как ошибки. Если есть веские причины их устранить, то инженеры должны это сделать, иначе сборка завершится неудачей. Если обнаруженные ошибки или предупреждения приводят к *ложным срабатываниям*, их не станут устранять — процесс их проигнорирует. В этом случае сборка будет выполнена успешно.

Следствием этого принципа является *политика нулевой толерантности* к предупреждениям. Если сборка и анализ проекта постоянно выявляют много

ошибок и предупреждений, пусть и некритических, — это источник возможных проблем. Наличие в проекте предупреждений и сообщений об ошибках не дает составить адекватное представление о его качестве. Инженеры не могут с первого взгляда определить, действительно ли сотни сообщений являются проблемой. Кроме того, множество сообщений о проблемах демотивирует инженеров — они перестают исправлять недавно появившиеся предупреждения. Проведем параллель — представим дом в ужасном состоянии: стены разрушены, окна разбиты. Никто не огорчится, если треснет стекло еще в одном окне. Но если будет разбито окно в ухоженном доме, это побудит хозяина немедленно принять меры. То же самое можно сказать и о проверке качества программного обеспечения. Если уже есть сотни предупреждений, то никто не станет беспокоиться о недочетах в последнем зафиксированном состоянии. Поэтому количество нарушений качества проекта должно быть равным нулю. Ошибки в сборках или при анализе кода должны прерывать конвейер сборки. Затем нужно либо исправить код проекта, либо скорректировать критерии качества, чтобы устранить проблему.

Инструменты контроля качества кода, такие как SonarQube, интегрируются в конвейер как отдельный этап сборки. Поскольку анализ качества работает только на статическом входе, эту операцию можно легко выполнять параллельно со следующими этапами конвейера. Если контроль качества не будет пройден, то сборка завершится неудачно и инженерам придется устранять проблему, прежде чем продолжить разработку. Это важный аспект интеграции контроля качества в конвейер сборки. Анализ должен не только давать представление о качестве проекта, но и активно препятствовать дальнейшим действиям в случае, если качество неудовлетворительное.

Развертывание

После сборки двоичных файлов и после или во время проверки качества программного обеспечения происходит развертывание корпоративного приложения. В зависимости от условий проекта для тестирования, как правило, используются несколько сред, таких как среды тестирования, переноса и, конечно же, эксплуатации. Как уже отмечалось, они должны быть как можно более схожими. Это значительно упрощает процесс развертывания под управлением сервера непрерывной интеграции.

Процесс развертывания приложения обычно берет двоичные файлы из только что собранной версии и развертывает их в среде выполнения. В зависимости от инфраструктуры это может происходить с использованием простых сценариев или более сложных технологий. Основной принцип всегда одинаков: двоичные файлы, а также конфигурация делаются доступными в среде выполнения надежным автоматическим способом. На этом этапе реализуются также подготовительные операции, которые, возможно, потребуются для приложения или среды выполнения.

Современные среды выполнения, такие как фреймворки управления контейнерами, поддерживают инфраструктуру как код. Конфигурация инфраструктуры записывается в файлах, хранится в репозитории проекта и применяется ко всем средам во время развертывания. Возможные различия, например в содержимом конфигурационных карт Kubernetes, также сохранены в репозитории в виде различных представлений.

Использование IaC, а также контейнеров обеспечивает бóльшую надежность, чем сценарии оболочки собственной разработки. Приложение всегда должно разворачиваться идемпотентным способом независимо от того, в каком состоянии находится среда выполнения. Поскольку образ контейнера содержит весь стек, то результат всегда будет таким, как если бы программное обеспечение устанавливалось с нуля. Необходимая конфигурация окружения также берется из файлов IaC.

Новые версии образов контейнеров могут быть развернуты разными способами с помощью фреймворков управления. Существуют определенные команды, которые явно устанавливают образы Docker, используемые в развертываниях Kubernetes. Но чтобы удовлетворять требованиям надежности и воспроизводимости, имеет смысл редактировать только файлы инфраструктуры как кода и применять их в кластере. Это гарантирует, что файлы конфигурации остаются единственным источником истины. Сервер непрерывной интеграции может редактировать определения образов в файлах IaC и фиксировать изменения в репозитории VCS.

Как следует из предыдущей главы, образы Docker указываются в определениях развертывания Kubernetes таким образом:

```
# описание развертывания как в главе 5
# …
    spec:
        containers:
        - name: hello-cloud
            image: docker.example.com/hello-cloud:1
            imagePullPolicy: IfNotPresent
            livenessProbe:
# …
```

Эти описания образов обновляются в процессе работы сервера непрерывной интеграции и применяются к кластеру Kubernetes. CI-сервер выполняет команды Kubernetes через интерфейс командной строки `kubectl`. Это стандартный способ коммуникации с кластерами Kubernetes. По команде `kubectl apply -f <file>` применяется содержимое инфраструктуры как кода, представленное в виде файла или каталога, хранящего определения YAML или JSON. На этапе конвейера выполняется команда, аналогичная этой, предоставляя файлы Kubernetes, которые были обновлены в репозитории проекта.

В соответствии с этим методом в файлах инфраструктуры как кода содержатся текущее состояние среды выполнения, а также изменения, внесенные инженерами. Все обновления развертываются путем использования в кластере соответствующей версии файлов Kubernetes. Кластер стремится прийти к новому

желаемому состоянию, содержащему новую версию образа, и поэтому проводит последовательное обновление. После запуска этого обновления сервер непрерывной интеграции проверяет, успешно ли прошло развертывание. После того как команды развертывания Kubernetes выполнены, могут поступить команды, подобные `kubectl rollout status <deployment>`, которые ожидают, пока завершится развертывание — удачно или нет.

Эта процедура выполняется во всех средах. Разумеется, если определения для одного развертывания используются в нескольких средах, то описание метки образа нужно обновить только один раз.

Более наглядный пример приведен на рис. 6.2 — представлена возможная структура файлов конфигурации проекта Maven.

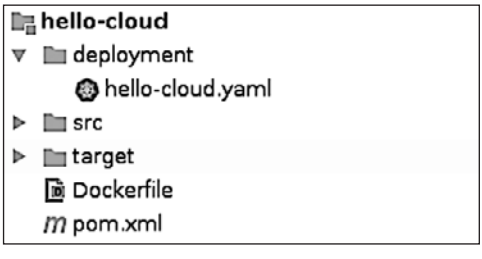

Рис. 6.2

В файле `hello-cloud.yaml` содержится несколько определений ресурсов Kubernetes. Для этого определения объектов YAML разделяют строкой, состоящей из трех дефисов (`---`). Взамен можно создать отдельные файлы для каждого типа ресурсов — `deployment.yaml`, `service.yaml` и т. д. Kubernetes поддерживает оба варианта. Тип ресурса в объектах YAML указывается с помощью определений типа `kind`.

В главе 5 было показано, как фреймворки управления контейнером позволяют проводить развертывания с нулевым простоем без специальной настройки. Применение новых версий образов в средах выполнения, управляемых сервером непрерывной интеграции, также способствует достижению этой цели. Таким образом, среда выполнения всегда будет способна обслуживать трафик, так как по меньшей мере одно приложение будет активно в каждый момент. Этот подход особенно важен для условий эксплуатации.

Конфигурация

В идеале инфраструктура как код охватывает все аспекты, необходимые для описания всей среды, включая период выполнения, сетевое взаимодействие и конфигурацию. Использование контейнерных технологий и управления контейнерами серьезно поддерживает и упрощает этот подход. Как уже отмечалось, конфиденциальные сведения, например учетные данные, не должны храниться в системе контроля версий. Администратор должен вводить их вручную.

Если конфигурация различается в разных средах выполнения, то в репозитории проекта она может быть представлена в виде нескольких файлов. Например, имеет смысл создать вложенные папки для каждой среды. Пример такой конфигурации показан на рис. 6.3.

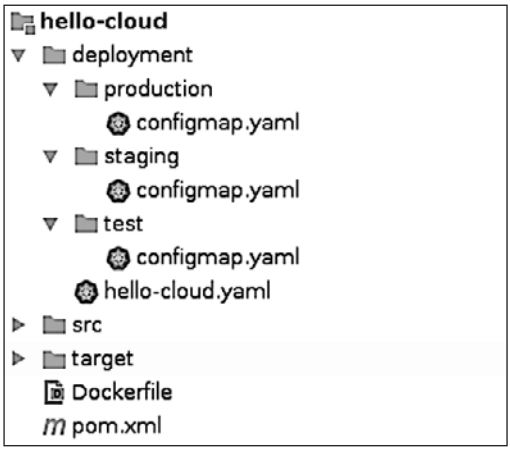

Рис. 6.3

В файле `configmap.yaml` хранится содержимое определенной конфигурационной карты, возможно, там же находятся различные определения пространства имен. Как отмечалось в главе 5, пространства имен Kubernetes — это способ различать среды выполнения. В следующем коде показан пример конфигурационной карты `production`:

```
---
kind: ConfigMap
apiVersion: v1
metadata:
    name: hello-cloud-config
    namespace: production
data:
    application.properties: |
        hello.greeting=Hello production
        hello.name=Java EE
---
```

Учетные данные

По соображениям безопасности секретный контент, такой как учетные данные, обычно не включается в репозиторий проекта. Как правило, администратор вводит его вручную в каждой из сред. Подобно другим ресурсам Kubernetes, секреты привязаны к определенному пространству имен.

Если в проекте предусмотрено множество секретных данных, например учетные данные для различных внешних систем, вводить их вручную может ока-

заться утомительной и трудно отслеживаемой операцией. Сконфигурированные секретные данные нужно внести в документацию, а отслеживать их необходимо в защищенной форме вне репозитория проекта.

Другой метод заключается в хранении зашифрованных учетных данных, которые могут быть расшифрованы с помощью единого главного ключа в репозитории. Поэтому в репозитории можно безопасно хранить сконфигурированные учетные данные в зашифрованном виде и не беспокоиться о том, что секреты будут раскрыты. После запуска приложение станет использовать динамически предоставляемый мастер-ключ для расшифровки сконфигурированных учетных данных. Такой подход обеспечивает безопасность и управляемость.

Рассмотрим возможный вариант решения. Зашифрованные значения конфигурации могут надежно храниться в конфигурационных картах Kubernetes, поскольку расшифрованные значения будут видны только в процессе контейнера. Зашифрованные учетные данные, как и другие значения конфигурации, могут быть представлены в проекте как определения конфигурационных карт в виде кода. Администратор добавляет для каждой среды выполнения секретную информацию о главном ключе, используемом для симметричного шифрования учетных данных. Этот ключ предоставляется работающему контейнеру, например, через переменные среды, как было показано ранее. Работающее приложение задействует эту единственную переменную среды для расшифровки всех зашифрованных значений учетных данных.

Одним из решений, зависящих от технологии и алгоритма, является использование приложения Java EE для расшифровки учетных данных непосредственно при загрузке файлов свойств. Чтобы обеспечить безопасное решение с применением новейших алгоритмов шифрования, в среде выполнения следует установить *Java Cryptographic Extensions (JCE)*. Другой подход заключается в расшифровке значений до развертывания приложения.

Миграция данных

Приложения, использующие базу данных для хранения своего состояния, привязаны к конкретной схеме базы данных. Изменения в схеме обычно требуют изменения модели приложения, и наоборот. Поскольку приложение активно развивается, а модель предметной области постоянно совершенствуется и реорганизуется, эта модель рано или поздно потребует изменения схемы базы данных. Новые классы моделей или новые добавляемые свойства необходимо сохранять в базе данных. Классы и свойства, которые подверглись реструктуризации или были изменены, также должны быть перенесены в базу данных, чтобы приложение не отличалось от схемы.

Провести миграцию данных сложнее, чем изменить код. Приложения без сохранения состояния могут быть просто заменены новыми версиями с новой функциональностью. Однако в базу данных, в которой хранится состояние приложения, необходимо тщательно перенести новое состояние при изменении схемы.

Делается это с помощью сценариев миграции. В реляционных базах данных можно изменять таблицы, сохраняя данные без изменений. Эти сценарии выполняются до развертывания новой версии программного обеспечения и следят за тем, чтобы схема базы данных соответствовала новому приложению.

Существует важная особенность, которую следует учитывать при развертывании приложений с использованием принципа нулевого простоя. При развертывании обновлений в среде в один момент будет работать как минимум один активный экземпляр приложения. Это приводит к тому, что в течение короткого периода времени будут активны и старая и новая версии приложения. Система управления должна позаботиться о том, чтобы первая из них была плавно закрыта, успев обработать текущие запросы, а вторая — так же плавно запущена. Если все приложения подключаются к центральному экземпляру базы данных, это приведет к тому, что к базе данных будут обращаться одновременно разные версии приложения. Требуется, чтобы приложение поддерживало так называемую *совместимость N – 1*: текущая версия приложения должна работать с той же версией схемы базы данных плюс-минус одна версия.

Для поддержки совместимости $N - 1$ при развертывании обновления необходимо установить новую версию приложения и обновить схему базы данных, гарантируя при этом, что они различаются не более чем на одну версию. Это означает, что соответствующая миграция базы данных выполняется непосредственно перед развертыванием приложения. Поэтому схема базы данных, как и приложение, переносится постепенно, небольшими шагами, а не рывком.

Такой подход нельзя назвать банальным. Он предполагает определенное планирование и осторожность. Особенно важно обратить внимание на откаты версий приложений.

Добавление структур баз данных

Добавить таблицу или отдельные столбцы таблицы в схему базы данных довольно просто. Новая таблица или столбец не входят в конфликт со старыми версиями приложений, поскольку те им неизвестны.

Новые таблицы для хранения новых сущностей предметной области могут быть просто добавлены в схему, что приведет к появлению версии $N + 1$.

При создании новых столбцов таблицы, имеющих некие ограничения (например, их значения должны быть `not null` или `unique`), следует позаботиться о текущем состоянии таблицы. Старая версия приложения все еще может производить запись в такую таблицу, при этом новый столбец будет проигнорирован и, следовательно, ограничения не будут выполнены. Новые столбцы сначала должны допускать *нулевые значения* и не иметь дополнительных ограничений. В новой версии приложения нужно учесть, что вначале этот столбец будет пустым, предположительно со значениями `null`, полученными из старой версии приложения.

Только в следующей версии ($N + 2$) можно будет после завершения текущего развертывания ввести правильные ограничения. Это означает, что для того, чтобы добавить столбец с ограничениями, требуется как минимум два отдельных раз-

вертывания: первое добавляет столбец и улучшает модель приложения с учетом возможных значений `null`, второе гарантирует, что все значения, содержащиеся в таблице, удовлетворяют ограничениям столбцов, добавляет ограничения и удаляет поведение, рассчитанное на значения `null`. Разумеется, эти этапы нужны только в том случае, если желаемый формат столбца содержит ограничения.

Откаты к старым версиям выполняются аналогично. При возвращении к промежуточному развертыванию (от $N + 2$ к $N + 1$) вновь требуется удалить ограничения.

При возврате в исходное состояние ($N + 0$) весь столбец будет удален. Однако при миграции данных не должны удаляться данные, если они не переданы куда-то еще. При возврате в состояние без столбца можно просто оставить столбец нетронутым, чтобы не потерять данные. Вопрос о том, что делать с данными, добавленными за это время, надо задать бизнес-специалистам. Возможно, удалять их не стоит. Однако, когда снова понадобится добавить столбец в приложение, нужно будет учесть в сценарии развертывания, что в базе данных он уже существует.

Изменение структуры базы данных

Изменение существующих таблиц или столбцов базы данных — более сложная операция. Независимо от того, что именно меняется, имя столбца, тип или ограничения, переход должен выполняться за несколько этапов. Прямое переименование или изменение столбцов приведет к несовместимости с уже развернутыми экземплярами приложений. Чтобы изменение было правильным, нужно создать промежуточные столбцы.

Рассмотрим этот метод на примере. Предположим, что объект «машина» имеет свойство «цвет», которое хранится в таблице базы данных, в столбце `color`. Предположим, что после реструктуризации приложения это свойство переименовано в «цвет кузова», соответственно столбец базы данных будет называться `chassis_color`.

Как и в предыдущем случае, изменение выполняется за несколько развертываний. При первом развертывании в таблицу добавляется столбец `chassis_color`, значения которого могут быть в том числе пустыми. Код приложения расширен с учетом использования нового свойства модели. Поскольку более старая версия приложения не знает о новом свойстве, применять его во время первого развертывания будет рискованно. Поэтому первая версия кода по-прежнему считывает цвет из старого столбца `color`, но записывает его значения и в старый, и в новый столбцы.

При следующем развертывании сценарий миграции обновляет недостающие значения нового столбца `chassis_color`, копируя туда содержимое столбца `color`. Таким образом обеспечивается заполнение нового столбца. Для него также вводится ограничение `not null`. Теперь версия кода приложения будет считывать данные только из нового столбца, но записывать данные по-прежнему в оба — и в старый, и в новый, поскольку в течение короткого периода старая версия все еще будет активна.

На следующем этапе развертывания удаляется ограничение `not null` для столбца `color`. В этой версии код приложения больше не использует старый столбец, чтение и запись выполняются только в `chassis_color`.

На следующем, последнем этапе развертывания столбец `color` удаляется. Теперь все данные переданы в новый столбец `chassis_color`. В коде приложения больше нет свойства «цвет» старой модели.

Изменение типа столбца или ограничений внешнего ключа требует аналогичных действий. Единственный способ постепенной миграции базы данных с нулевым временем простоя — это поэтапная миграция с использованием промежуточных столбцов и свойств. Рекомендуется несколько раз зафиксировать изменения, содержащие только эти изменения сценариев миграции и кода приложения.

Как и в предыдущем случае, миграция отката должна протекать в обратном порядке как для сценариев базы данных, так и для изменения кода.

Удаление структур базы данных

Удалять таблицы или столбцы проще, чем изменять их. Если какие-либо свойства модели предметной области больше не требуются, то сценарии их использования можно удалить из приложения.

На первом этапе развертывания изменяют код приложения, чтобы прекратить чтение из столбца базы данных, но сохраняют запись в него. Это необходимо, чтобы старая версия все еще могла читать значения, отличные от `null`.

На следующем этапе развертывание удалит ограничение `not null` для этого столбца базы данных (если оно было). Код приложения прекращает запись в столбец. На этом этапе свойство модели уже можно удалить из базы кода.

На заключительном этапе развертывания столбец будет удален. Как уже отмечалось, то, действительно ли нужно удалить данные из этого столбца, сильно зависит от бизнес-сценария. В случае отката удаленный столбец потребуется восстановить, что подразумевает, что его содержимое было утрачено.

Реализация миграции

Как видим, миграцию данных нужно выполнять в несколько этапов. Сценарии установки и отката реализуются непосредственно перед развертыванием. Это означает, что приложение поддерживает совместимость $N - 1$, а также то, что развертывания проходят последовательно, по одному.

Процесс миграции требует выполнения нескольких версий программного обеспечения, каждая из которых состоит из согласованного кода приложения и сценариев миграции данных. Инженеры должны планировать соответствующие фиксации изменений проекта. Желательно произвести полную миграцию схемы своевременно, чтобы сохранить чистую схему базы данных и гарантировать, что текущие миграции не будут просто забыты.

Вопрос о том, нужно ли сохранить существующие данные или их можно удалить, определяется выбранной моделью реструктуризации. Вообще говоря,

желательно не выбрасывать данные. Это означает, что не следует удалять содержащие их структуры, если эти данные не хранятся еще где-нибудь.

Как мы видели в примерах, миграция будет выполняться постепенно, мелкими шагами. Особенно важно это для ограничений базы данных, таких как ненулевые значения или ограничения целостности. Сценарии миграции должны быть стабильными. Например, миграция не должна прерываться при попытке создания уже существующих столбцов. Они могли остаться после предыдущих откатов. Вообще говоря, имеет смысл продумать и протестировать различные сценарии развертывания и отката.

При обновлении содержимого таблиц инженерам следует помнить о времени обновления. Обновление очень больших таблиц — длительный процесс, в течение которого данные будут заблокированы. Это нужно учитывать заранее — в идеале путем тестирования сценариев обновления в отдельной базе данных. При очень большом количестве задействованных данных этапы обновления могут выполняться по частям, например посредством разбиения данных по идентификаторам.

Все сценарии развертывания и отката миграции должны храниться в репозитории проекта. Схема базы данных содержит номер версии, соответствующий версии сценария миграции. Последняя хранится в базе данных как метаданные вместе с текущим состоянием схемы. Перед каждым развертыванием схема базы данных мигрирует до желаемой версии. Сразу после этого развертывается соответствующая версия приложения. При этом необходимо проследить, чтобы версии не различались более чем на единицу.

Для фреймворка управления контейнером это означает, что миграция базы данных должна выполняться методом скользящих обновлений непосредственно перед развертыванием новой версии приложения. Поскольку может существовать много реплик модулей, этот процесс должен быть идемпотентным. Повторная миграция той же версии схемы базы данных всегда должна давать неизменный результат. Модули Kubernetes могут определять так называемые *контейнеры инициализации*, выполняющие однократные процессы перед запуском основного контейнера. Контейнеры инициализации являются взаимоисключающими. Они должны успешно завершить работу до того, как запустится процесс модуля контейнеров.

В следующем фрагменте кода показан пример контейнера инициализации `initContainer`:

```
# …
    spec:
        containers:
        - name: hello-cloud
            image: .../hello-cloud:1
        initContainers:
        - name: migrate-vehicle-db
            image: postgres
            command: ['/migrate.sh', '$VERSION']
# …
```

В этом примере подразумевается, что образ контейнера инициализации содержит все необходимые инструменты для подключения к экземпляру базы

данных, а также ко всем последним версиям сценариев миграции. Для того чтобы это стало возможным, образ также собирается как часть конвейера, включая все сценарии миграции из репозитория.

Есть много решений для переноса схем баз данных. Важной особенностью является то, что идемпотентная миграция должна быть проделана заранее и в это время не должно протекать никаких других операций развертывания. Сценарии миграции соответствующих версий должны выполняться в порядке возрастания или убывания в зависимости от того, что реализуется, обновление или откат схемы базы данных до нужной версии. После того как сценарии будут выполнены, версия метаданных в базе данных также обновляется.

Соответствие между версиями кода и базы данных можно отслеживать в репозитории проекта. Например, самый последний сценарий развертывания, содержащийся в версии фиксации изменений, соответствует требуемой схеме базы данных. Подробнее о том, какие метаданные нужны и где их хранить, читайте в подразделе «Метаданные сборки».

Поскольку выбор решения для миграции сильно зависит от технологии проекта, не существует «универсальной таблетки» на все случаи жизни, которую можно было бы здесь привести. В следующем примере показан один из возможных вариантов файловой структуры и выполнения миграции на псевдокоде (рис. 6.4). Здесь приводятся файлы миграции для обсуждавшегося ранее примера с изменением столбца `color` на `chassis_color`.

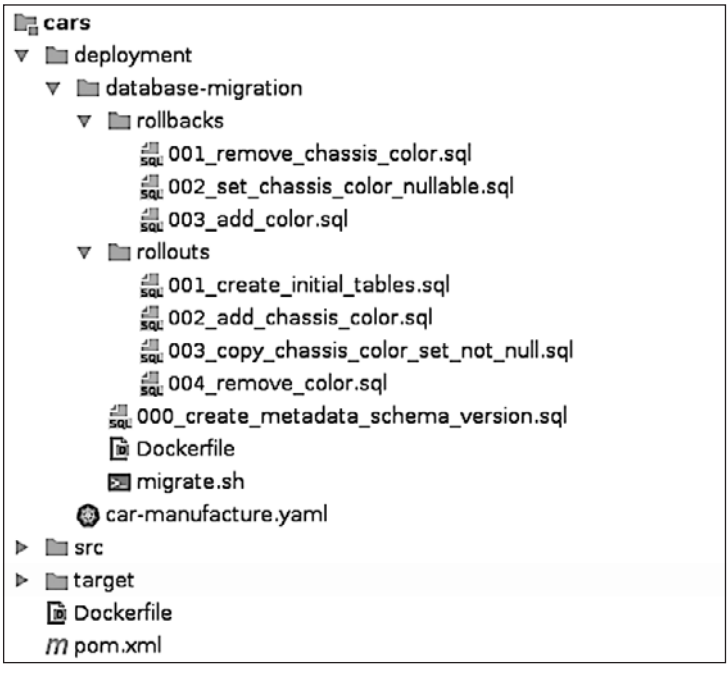

Рис. 6.4

В предыдущем примере были показаны сценарии развертывания и отката, которые переводят версию схемы базы данных в желаемое состояние. Сценарий развертывания `004_remove_color.sql` переводит схему в версию 4, удаляя столбец `color` в приведенном ранее примере. Соответствующий сценарий отката `003_add_color.sql` откатывает схему до версии 3, где столбец `color` все еще существует. Другими словами, версия 3 содержит столбец `color`, тогда как в версии 4 его уже нет, причем оба эти файла миграции позволяют совершать как развертывание, так и откат.

Далее показан псевдокод сценария, выполняющего миграцию. Желаемая версия миграции предоставляется при вызове сценария в качестве аргумента:

```
текущая_версия = текущая версия схемы, хранящаяся в базе данных

если текущая_версия == новая_версия
    выйти, ничего не делать

если текущая_версия < новая_версия
    папка = /rollouts/
    последовательность_сценариев = последовательность от текущая_версия + 1
        до новая_версия

if текущая_версия > новая_версия
    папка = /rollbacks/
    последовательность_сценариев = последовательность от текущая_версия - 1
        до новая_версия

for i in последовательность_сценариев
    выполнить сценарий в папка/i_*.sql
    обновить версию схемы до i
```

Этот сценарий миграции выполняется в контейнере инициализации перед развертыванием.

Тестирование

Проверка результата, получаемого после каждого этапа конвейера, — одна из самых важных составляющих непрерывной поставки. Она повышает качество программного обеспечения, позволяя обнаружить возможные ошибки, прежде чем приложение поступит в эксплуатацию. Надлежащая проверка повышает надежность процессов. Создавая программные тесты в целом и регрессионные тесты в частности, разработчики гарантируют работоспособность функций приложения после их изменения и реструктуризации. В конечном счете тесты программного обеспечения позволяют автоматизировать процессы разработки.

Тесты на уровне кода выполняются уже при сборке двоичных файлов. Другие тесты, содержащиеся в проекте, могут реализовываться на отдельных этапах конвейера. Какими будут эти этапы, зависит от того, работают ли они на уровне кода или в действующем контейнере. В частности, для сквозных тестов требуется работающая среда.

После развертывания приложения в тестовых средах в них могут быть выполнены сквозные тесты. Обычно проект содержит несколько уровней тестов со своими задачами на каждом этапе. В зависимости от проекта и применяемой технологии может использоваться множество различных тестов. Главное — на каждом этапе конвейера тщательно проверять полученный результат. При этом минимизируется риск нарушения новых или существующих функциональных возможностей и появления ошибок. В частности, фреймворки управления контейнерами с реализованным в них принципом *готовности к эксплуатации* поддерживают компании в их стремлении поставлять масштабируемые, надежные корпоративные приложения высокого качества. Подробнее о различных вариантах тестирования, включая выполнение тестов в конвейерах непрерывной поставки, читайте в главе 7.

Неудачное прохождение тестов должно вызывать немедленную остановку конвейера и предотвращать дальнейшее использование соответствующего двоичного файла. Этот важный принцип обеспечивает быструю обратную связь, а также высокое качество программного обеспечения в процессе сборки. Инженерам не следует пропускать стандартные этапы процесса и практиковать другие способы решить проблему на скорую руку. Это противоречит идее непрерывного совершенствования и обеспечения качества сборки в процессе непрерывной поставки и в итоге приводит к появлению ошибок. Если приложение не проходит тесты или проверку качества, то сборка должна быть прервана и либо исправлен код приложения, либо изменены условия проверки.

Неудачное прохождение тестов должно не только прерывать сборку, но и давать представление о том, почему эта операция не удалась, и сохранять результат как часть метаданных сборки.

Метаданные сборки

В метаданные сборки записывается вся информация, полученная во время ее выполнения. В частности, это версии всех объектов, которые нужно отслеживать для дальнейшего использования.

Если сборка каждый раз выполняется с начала до конца, то дополнительная информация не нужна. Все этапы реализуются за один проход до тех пор, пока сборка не завершится успешно или не будет прервана. Если же определенные этапы или артефакты должны быть выполнены повторно или на них нужно сослаться, то требуется дополнительная информация.

Ярким примером такой необходимости являются версии артефактов. WAR-файл и его содержимое соответствуют определенной версии в истории фиксаций изменений в системе контроля версий. Для того чтобы отслеживать исходную версию из развернутого приложения, эту информацию нужно где-то хранить. То же самое касается и версий образов контейнеров. Для того чтобы идентифицировать происхождение и содержимое контейнера, нужно иметь возможность отслеживать версии. Еще один пример — версии схемы базы данных. Каждая такая версия соответствует конкретной версии приложения, включая предыдущую и следующую версии, в соответствии с правилом совместимости $N - 1$.

Для развертывания с миграцией схемы базы данных необходимо знать версию схемы, до которой нужно мигрировать, для данной версии приложения.

Метаданные сборки особенно необходимы, когда процесс допускает развертывание конкретных версий приложения. Обычно при непрерывной поставке развертывается текущая версия из репозитория. Однако возможность развертывания среды выполнения в произвольное состояние является огромным преимуществом, особенно при использовании схем и миграций баз данных. Теоретически процесс работает следующим образом: *взять конкретную версию приложения и проделать все необходимое для ее запуска в конкретной среде* независимо от того, является это переходом к следующей или откатом к предыдущей версии.

Для того чтобы лучше отслеживать и повышать воспроизводимость, желательно также контролировать информацию о качестве сборки. К ней относятся, например, результаты автоматических и ручных тестов, а также анализа качества кода. Затем можно проверить наличие определенных метаданных на соответствующих этапах перед развертыванием.

Есть множество решений для представления метаданных. Некоторые репозитории артефактов, такие как JFrog Artifactory, обеспечивают возможность связывания собранных артефактов с пользовательскими метаданными.

Другой вариант — использовать CI-сервер для отслеживания этой информации. Это звучит неплохо для хранения метаданных сборки, однако в зависимости от того, как работает и настраивается сервер непрерывной интеграции, иногда не рекомендуется хранить на нам постоянные данные. Старые сборки могут отбрасываться, и информация о них может быть утрачена.

В целом количество *точек истины*, например, для хранения артефактов и информации должно быть небольшим и четко определенным. Поэтому имеет смысл использовать репозитории артефактов для метаданных.

Другое, более индивидуальное решение — задействовать VCS-репозитории компании для отслеживания определенной информации. Большим преимуществом хранения метаданных, например, в Git является то, что при этом обеспечивается полная гибкость сохраняемых данных и структуры. Серверы непрерывной интеграции по умолчанию обладают функциональными возможностями для доступа к VCS-репозиториям, поэтому инструменты сторонних производителей не требуются. Репозитории позволяют хранить любую информацию, которая может быть представлена в виде файлов, например записанные результаты тестирования.

Реализованный репозиторий метаданных доступен в разных точках конвейера, например в процессе развертывания.

Передача в эксплуатацию

Последний этап конвейера непрерывной поставки — это развертывание в среде эксплуатации. Оно запускается вручную или, после того как пройдены все проверки и автоматические тесты, автоматически. Подавляющее большинство компаний используют развертывание с ручным запуском. Но даже если конвейер не был

пройден с начала и до конца, непрерывная поставка дает большие преимущества, автоматизируя все необходимые операции.

В конвейере есть только две стартовые точки: начальная фиксация изменений в репозитории, которая запускает выполнение, и окончательное развертывание для эксплуатации после того, как все этапы были проверены вручную и автоматически.

В среде управления контейнерами развертывание для эксплуатации, то есть развертывание в отдельном пространстве имен или отдельном кластере, происходит так же, как и развертывание в тестовых средах. Поскольку инфраструктура как код аналогична или идентична тем, которые выполнялись ранее, эта технология снижает риск несоответствия среды для эксплуатации.

Модели ветвления

В ходе разработки программного обеспечения удобно использовать различные модели ветвления. Ветви программного обеспечения возникают из общего исходного варианта и различаются на определенном этапе, что позволяет параллельно вести несколько этапов разработки.

Особенно распространено ветвление функций. При этом создаются отдельные ветви для разработки каждой функции программного продукта. При завершении работы над функцией ветвь объединяется с главной ветвью. Пока функция разрабатывается, главная ветвь разработки и другие ветви остаются нетронутыми.

Еще одна модель ветвления — использование ветвей конечных версий. Ветви конечных версий содержат готовое программное обеспечение определенных версий. Суть в том, чтобы создать определенную точку для выпущенной версии, где к ней могут добавляться исправления ошибок и новые функции. Все внесенные в главную ветвь изменения, которые применяются для конкретной версии, выполняются и в ветви версии.

Однако ветвящиеся модели противоречат идее непрерывной поставки. Например, ветви функций задерживают интеграцию этих функций в главную ветвь. Чем дольше новые функции не интегрируются, тем больше вероятность конфликтов слияния. Таким образом, ветви функций рекомендуется использовать в конвейерах непрерывной поставки только в том случае, если они недолговечны и своевременно интегрируются в главную ветвь.

Выпуск разных версий и параллельная работа над ними противоречат идее непрерывной поставки версий. Реализованные функции в идеале должны поставляться в среду эксплуатации как можно быстрее.

Это относится по крайней мере к корпоративным проектам. Продолжительный жизненный цикл подразумевает, что каждая фиксация изменений — потенциальный кандидат для развертывания в среде эксплуатации. Имеет смысл интегрировать и использовать в рабочей среде главную ветвь, чтобы как можно раньше интегрировать и развертывать функции, проверенные автоматическими тестами. Таким образом, модель ветвления при непрерывных поставке и развер-

тывании является довольно простой. Изменения применяются непосредственно к главной ветви, собираются, проверяются и развертываются конвейером сборки.

Добавлять метки вручную обычно не требуется. Каждая фиксация изменений в конвейере непрерывной поставки неявно квалифицируется для выпуска и развертывания в среде эксплуатации, если только не обнаружатся ошибки в ходе автоматической проверки.

На рис. 6.5 показана концепция модели ветвления при непрерывном развертывании.

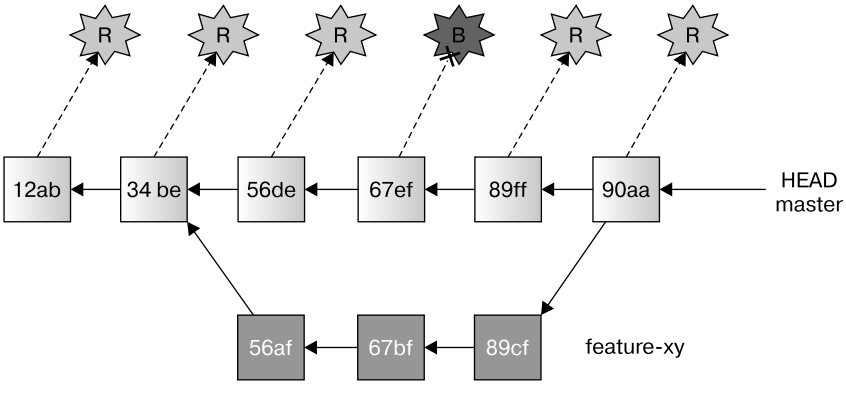

Рис. 6.5

Отдельные ветви функций существуют недолго и своевременно сливаются с главной ветвью. Выпуски создаются неявно при успешных сборках. Прерванные сборки не приводят к развертыванию в среде эксплуатации.

Однако продукты и библиотеки могут иметь разные модели ветвления. Если существует несколько *основных* и *второстепенных* версий и в каждой из них ведется работа по исправлению ошибок, то имеет смысл создавать отдельные ветви для каждой такой версии. Ветви окончательной версии, такие как v1.0.2, можно впоследствии использовать для технической поддержки и исправлений ошибок, продолжая нумерацию, например, как v1.0.3. Основная разработка в это время продолжается в более новой ветви с номером, например, v2.2.0.

Технология

При проектировании конвейеров непрерывной поставки возникает вопрос, какую технологию использовать. Речь идет не только о самом CI-сервере, но и обо всех инструментах, применяемых в ходе разработки, таких как система управления версиями, хранилища артефактов, системы сборки и технологии среды выполнения.

Выбор технологии зависит от реальных потребностей и не в последнюю очередь от того, с какими технологиями знакома группа инженеров. В следующих примерах будут применены Jenkins, Git, Maven, Docker и Kubernetes, широко

распространенные во время написания этой книги. Однако для инженеров важнее понимать основополагающие принципы и мотивацию. Сами же технологии взаимозаменяемы.

Независимо от того, какие именно инструменты будут выбраны, рекомендуется использовать их по прямому назначению. Опыт показывает, что инструменты часто применяются для решения задач, для которых они не предназначены и которые лучше выполнить, задействуя другие технологии. Ярким примером является система сборки Maven. В проектах часто создаются процессы сборки, которые наряду с созданием артефактов решают дополнительные задачи.

Имеет смысл не смешивать операции по созданию контейнеров или развертыванию программного обеспечения при сборке артефактов. Эти задачи предпочтительно реализовать непосредственно на сервере непрерывной интеграции. Перенос данных операций в процесс сборки связывает технологию сборки со средой выполнения, хотя в этом нет необходимости.

Итак, рекомендуется использовать инструменты для того, для чего они непосредственно предназначены. Например, стоит строить контейнеры Docker из соответствующих двоичных файлов Docker, а не создавать дополнительные системные модули. Необходимые уровни абстракции лучше добавлять в конвейер как определения кода, как будет показано в следующих примерах.

Конвейер как код

Мы уже видели преимущества представления конфигурации как кода, прежде всего инфраструктуры как кода в виде файлов. Те же причины привели нас к определению конвейера как кода — конфигурации, определяющей этапы конвейера CI-сервера.

В прошлом многие серверы непрерывной интеграции, такие как Jenkins, приходилось настраивать вручную. Все задачи CI-сервера тщательно объединялись в конвейерную последовательность. Поэтому перестройка конвейера с учетом нового приложения или новых ветвей функций требовала сложной ручной работы.

Определение конвейера как кода позволяет сделать конвейер непрерывной поставки частью программного проекта. CI-сервер создает и выполняет конвейер соответствующим образом, следуя сценарию. Это значительно упрощает определение и повторное использование проектов построения конвейеров.

Существует много серверов непрерывной поставки, которые поддерживают определение конвейера как кода. Самое важное заключается в том, что инженеры должны понимать причины применения этой технологии и знать ее преимущества. Далее приведены примеры для Jenkins — широко используемого в экосистеме Java CI-сервера.

Пользователи Jenkins могут создавать конвейеры в файле `Jenkinsfile`, который определяется с помощью Groovy DSL. Groovy — это опционально типизированный динамический JVM-язык, который хорошо подходит для создания DSL и сценариев. В сценариях сборки Gradle используется также Groovy DSL.

В следующих примерах показаны этапы очень простого конвейера для проекта Java. Они призваны дать приблизительное представление о выполненном процессе. Для получения полной информации о файлах Jenkinsfile, их синтаксисе и семантике советуем обратиться к документации.

Далее показан `Jenkinsfile`, содержащий определение базового конвейера:

```
node {
    prepare()

    stage('build') {
        build()
    }

    parallel failFast: false,
            'integration-test': {
                stage('integration-test') {
                    integrationTest()
                }
            },
            'analysis': {
                stage('analysis') {
                    analysis()
                }
            }

    stage('system-test') {
        systemTest()
    }

    stage('performance-test') {
        performanceTest()
    }

    stage('deploy') {
        deployProduction()
    }
}
// определения метода
```

Определения `stage` соответствуют этапам конвейера Jenkins. Поскольку сценарий Groovy представляет собой полноценный язык программирования, можно и нужно применять методы чистого кода, чтобы получить хорошо читаемый код. Поэтому содержимое конкретных этапов представлено в виде отдельных методов, которые находятся на одном уровне абстракции.

Например, на этапе `prepare()` инкапсулируются несколько методов, выполняющих предварительные условия сборки, таких как проверка репозитория сборки. В следующем коде показано определение этого метода:

```
def prepare() {
    deleteCachedDirs()
    checkoutGitRepos()
    prepareMetaInfo()
}
```

На этапе сборки также инкапсулируются несколько подэтапов, от выполнения сборки Maven, записи метаданных и результатов теста до создания образов Docker. В следующем коде показано определение метода:

```
def build() {
    buildMaven()
    testReports()
    publishArtifact()
    addBuildMetaInfo()
    buildPushDocker(dockerImage, 'cars')
    buildPushDocker(databaseMigrationDockerImage,
'cars/deployment/database-migration')
    addDockerMetaInfo()
}
```

Эти примеры дают представление о том, как описывать и инкапсулировать определенное поведение в виде этапов конвейера. Рассмотрение подробных примеров Jenkinsfile выходит за рамки книги. Я покажу только основные операции, позволяющие составить представление о том, какие логические действия необходимы и как их описать в сценариях конвейера понятным и эффективным способом. Однако фактические реализации сильно зависят от конкретного проекта.

Описания конвейера Jenkins позволяют подключать так называемые конвейерные библиотеки. Это готовые библиотеки, которые содержат часто используемые функции, упрощающие их применение и уменьшающие дублирование при наличии нескольких проектов. Стоит создать библиотеку компании и вынести туда определенные функции, особенно те, что касаются специфики конкретной среды.

В следующем примере показано развертывание приложения для производства автомобилей в среде Kubernetes. Метод `deploy()` вызывается из конвейера сборки при развертывании конкретного образа и схемы базы данных в пространстве имен Kubernetes:

```
def deploy(String namespace, String dockerImage, String databaseVersion) {
    echo "развертывание $dockerImage в Kubernetes $namespace"

    updateDeploymentImages(dockerImage, namespace, databaseVersion)
    applyDeployment(namespace)
    watchRollout(namespace)
}

def updateDeploymentImages(String dockerImage, String namespace, String
databaseVersion) {
    updateImage(dockerImage, 'cars/deployment/$namespace/*.yaml')
    updateDatabaseVersion(databaseVersion
'cars/deployment/$namespace/*.yaml')

    dir('cars') {
        commitPush("[jenkins] updated $namespace image to $dockerImage" +
            " and database version $databaseVersion")
    }
}

def applyDeployment(namespace) {
```

```
    sh "kubectl apply --namespace=$namespace -f
        car-manufacture/deployment/$namespace/"
}

def watchRollout(namespace) {
    sh "kubectl rollout status --namespace=$namespace deployments
        car-manufacture"
}
```

В этом примере определения Kubernetes YAML обновляются и сохраняются в VCS-репозитории в виде фиксации изменений. Все операции представлены в виде инфраструктуры как кода в пространстве имен Kubernetes и ожидают завершения развертывания.

Цель, которую я ставил, приводя эти примеры, — дать читателю представление о том, как интегрировать конвейеры непрерывной поставки в формате «конвейер как код» с фреймворком управления контейнерами, таким как Kubernetes. Как уже отмечалось, можно также задействовать библиотеки конвейеров для инкапсуляции часто используемых команд оболочки `kubectl`. Динамические языки, такие как Groovy, позволяют инженерам писать хорошо читаемые сценарии конвейера, а также любой другой код.

Рабочие процессы в Java EE

В приведенных примерах рассмотрены типичные конвейеры сборки Java, которые, конечно же, применимы и к Java EE. В сущности, Java Enterprise позволяет эффективно разрабатывать конвейеры. Быстрая сборка и, следовательно, быстрая обратная связь с программистами имеют решающее значение для построения эффективных рабочих процессов непрерывной поставки.

Как мы видели в главе 4, приложения с нулевой зависимостью, особенно упакованные в контейнеры, еще эффективнее используют эти принципы. Корпоративное приложение, представленное в виде упакованного артефакта или на уровне контейнера, содержит только бизнес-логику, разработанную на базе API. Ее реализацию обеспечивает контейнер приложений.

Конвейер непрерывной поставки выигрывает от использования приложений с нулевой зависимостью, поскольку задействованные этапы сборки и распространения требуют немного времени на выполнение и передачу. Сборка артефактов и контейнеров протекает максимально быстро, копируются только абсолютно необходимые элементы. Аналогично при публикации и развертывании артефактов и контейнеров выполняются только необходимые бизнес-задачи, что сокращает время передачи. Это позволяет ускорить цикл производства и обеспечивает быструю обратную связь.

Эффективные конвейеры критически важны для внедрения культуры непрерывной поставки в командах, где работают программисты. Это мотивирует инженеров выполнять тестирование как можно раньше и чаще, поскольку конвейер работает быстро, обеспечивает скорую обратную связь и повышает уверенность в том, что качество программного обеспечения находится на должном уровне.

Как уже говорилось, время сборки не должно превышать нескольких секунд. Прохождение всех этапов конвейера сборки, включая сквозные тесты, не должно занимать больше нескольких минут, в идеале еще меньше.

Разработчики должны прилагать максимум усилий к тому, чтобы сборки и конвейеры выполнялись как можно быстрее. В течение рабочего дня программисты часто успевают собрать и протестировать проект. В режиме непрерывной поставки каждая протестированная сборка является потенциальным кандидатом для развертывания в среде эксплуатации. Если весь процесс занимает, например, всего лишь на одну минуту больше, то все программисты в команде ждут на одну минуту дольше каждый раз, когда собирают программное обеспечение. Легко представить, что со временем эта задержка накапливается и становится довольно большой. Если инженерам приходится дольше ожидать результата, то у них появляется соблазн реже проводить тесты.

Таким образом, повышение стабильности и производительности конвейера — это долгосрочная инвестиция в продуктивность команды разработчиков. Тестирование и этапы, которые обеспечивают быструю и полезную обратную связь и срочно прерывают сборку в случае ошибок, должны выполняться как можно раньше. Если некоторые сквозные тесты занимают много времени и это нельзя изменить из-за характера проекта и самих тестов, то их можно описать как отдельные последующие этапы конвейера, чтобы не задерживать обратную связь после предыдущей проверки. Операции, которые могут протекать параллельно, например статический анализ кода, должны проходить так, чтобы сократилась общая продолжительность процесса. Современные подходы к разработке в Java EE позволяют создавать еще более эффективные контейнеры сборки.

Тем не менее технология — это лишь одна сторона построения эффективной непрерывной поставки. Внедрение непрерывной поставки еще сильнее влияет на культуру команды разработчиков. Рассмотрим этот аспект подробнее.

Культура непрерывной поставки и культура разработки

Эффективная непрерывная поставка зависит от грамотной культуры разработки. Если команда инженеров-программистов не действует по принципам и рекомендациям сообщества Continuous Delivery, то даже от самая лучшая технология не принесет пользы. Конвейеры, позволяющие реализовать автоматизированное развертывание, бесполезны, если нет достаточного количества программных тестов для проверки развернутого программного обеспечения. Самый мощный CI-сервер не поможет, если программисты редко проверяют вносимые изменения, из-за чего интеграция будет сложной и громоздкой. Полное покрытие тестами и проверка качества кода ничего не значат, если команда не реагирует на неудачные тесты или, что еще хуже, отключает их выполнение.

Ответственность

Непрерывная поставка начинается с ответственности за программное обеспечение. Как уже говорилось, в культуре DevOps недостаточно, чтобы программисты просто писали программы и предоставляли другим справляться с возможными ошибками. Команда программистов, которая создает приложение и владеет им, осознает свои обязанности, знает об используемых технологиях и устраняет неполадки, если возникнут ошибки.

Представьте себе небольшую компанию, в которой только один программист отвечает за приложение. Он занимается всеми техническими задачами — разработкой, сборкой, развертыванием и устранением неполадок в приложении. Он лучше всех знает все особенности приложения и может эффективно справиться с возможными проблемами. Очевидно, что такой подход с единой точкой ответственности не допускает масштабирования и приемлем только в крошечных компаниях.

На крупных предприятиях больше приложений, больше программистов и больше команд с разными обязанностями. Задача с разделением и переносом обязанностей заключается в передаче знаний. В идеале знания распространяются в команде инженеров, которые тесно сотрудничают в ходе работы над одним и тем же программным обеспечением. Как и в небольших начинающих компаниях, мантра для разработки приложений должна звучать так: *«Вы сами это создали, сами и запускайте»*. В единой команде это возможно только при наличии основных четко определенных автоматизированных процессов. Конвейеры непрерывной поставки позволяют реализовать эти процессы надежной поставки программного обеспечения.

Управление этими процессами и их совершенствование больше не являются проблемами команды эксплуатации, а становятся обязанностью всей команды инженеров. Все программисты несут равную ответственность за сборку и поставку работающего программного обеспечения, приносящего пользу бизнесу. Это, безусловно, требует выполнения определенных обязанностей и корпоративной культуры.

Проверять рано и часто

Непрерывной поставкой должна заниматься вся команда. Программисты, работающие над отдельными функциями или исправлением ошибок, должны как можно раньше и чаще фиксировать вносимые изменения в главной ветви. Это критически важно для обеспечения непрерывной интеграции. Чем больше времени проходит до того, как изменения объединятся с главной ветвью, тем сложнее выполнить слияние и интеграцию функций. Добавление сложной функциональности в формате «большой взрыв» противоречит идее *непрерывной* эволюции программного обеспечения. Функциональность, которая не должна быть видна пользователям, можно исключить переключением функций.

Необходимость слияния с главной ветвью часто побуждает разработчиков с самого начала писать достаточное количество автоматизированных тестов программного обеспечения. На это, безусловно, затратчивается время в ходе разработки, но затраты всегда будут окупаться в долгосрочной перспективе. Разрабатывая функцию, инженеры знают о ее задачах и ограничениях. Гораздо проще с самого начала написать не только модульные, но и сложные сквозные тесты, а не делать это после того, как функция будет написана.

Особенно важно сообщить менее опытным программистам, что выполнять слияние ветви разработки функций с главной ветвью на ранних этапах — не оплошность, а часть рабочего процесса. Код, который еще не реструктурирован и не выглядит идеальным, но выполняет требования и приносит пользу бизнесу, может быть очищен на следующем этапе. Гораздо важнее записать его в главную ветвь на ранней стадии процесса, чем ждать до последней минуты.

Проблемы немедленных исправлений

Немедленное устранение ошибок сборки — еще одна важная часть командной культуры разработки. Невозможно игнорировать или откладывать тесты, если приложение их не проходит, — следует исправить ошибки как можно быстрее. Если сборки часто прерываются из-за ошибок, но ничего не делается, чтобы решить проблему, то снижается продуктивность всех членов команды. Например, неудачный тест, из-за которого стала невозможной сборка проекта, не позволяет другим программистам интегрировать и проверять свои функции. Тем не менее неудачные сборки, возникшие из-за ошибок тестирования или недостаточного качества, являются признаком того, что валидация работает. И это, очевидно, гораздо лучше, чем ложные сообщения об отсутствии ошибок, то есть ошибочно успешные сборки. Тем не менее важно исправить сборку проекта сразу же, как только станет ясно, что она завершилась неудачно. Программисты должны выполнять основные быстрые проверки, такие как сборка проекта Java и тестирование на уровне кода, на локальных машинах перед тем, как записать проект в центральном репозитории. Следует позаботиться о том, чтобы не перегружать контейнер, вовремя обнаруживать случайные ошибки и не беспокоить лишний раз других членов команды.

Как уже говорилось, предупреждения компилятора и анализатора кода следует рассматривать как ошибки, прерывающие сборку. Это означает политику нулевой толерантности к предупреждениям, которая побуждает инженеров либо исправить ошибку, либо изменить условия проверки. Таким образом, предупреждения о сборке, компиляции или стиле кода также считаются ошибками, которые прерывают сборку и должны быть исправлены как можно скорее.

Член команды, который зафиксировал изменение, вызвавшее прерывание сборки, в идеале должен первым взяться за устранение причины ошибки. Однако ответственность за поддержание конвейера в работоспособном состоянии несет вся команда, поскольку она отвечает за весь проект. Ни у кого не должно

быть *исключительного права на использование кода*, то есть не должны существовать части проектов, которые известны только одному члену команды. Программисты, написавшие конкретную функциональность, всегда знакомы с ней лучше, чем остальные. Однако в любом случае вся команда должна иметь возможность работать со всеми частями проекта и при необходимости устранять проблемы.

Прозрачность

Еще одним важным показателем является прозрачность, которую обеспечивает непрерывная поставка. Можно весь процесс разработки, включая фиксацию изменений, сборку, проверки и развертывание, отслеживать из одной точки и составить общее представление о проекте. Какие характеристики прозрачности важны в конвейере непрерывной поставки?

Прежде всего необходимо указать, готово ли программное обеспечение к поставке. Сюда относится состояние сборки с точки зрения компиляции, тестов и анализа кода. Для того чтобы составить общее представление о ходе разработки, можно использовать панель мониторинга или так называемое устройство *экстремальной обратной связи* — табло с реальными зелеными и красными лампочками.

Разумная прозрачность в идеале не перегружает информацией при успешном выполнении сборки (зеленый свет), но в случае сбоев обеспечивает четкое и ясное понимание причин. Это снова приводит нас к принципу, что при сборке не бывает предупреждений — она либо проходит успешно и ничего делать не надо, либо прерывается и требует исправлений. Панели мониторинга или другие индикаторы красного или зеленого света уже сами по себе предоставляют полезные сведения. Эти инструменты прозрачности должны быть доступны всем членам команды для обеспечения совместной работы.

Но чтобы не слишком нарушать ежедневный процесс разработки, имеет смысл в первую очередь сообщать об ошибках тем, чьи фиксации изменений вызвали сбой при сборке. Скорее всего, они лучше знают, как исправить сборку, не прерывая без крайней необходимости работу коллег. Серверы непрерывной интеграции предоставляют функциональные возможности для отправки электронных писем, обмена сообщениями в чатах или других форм уведомления. Это повышает качество программного обеспечения, а также продуктивность работы программистов.

Информация, собранная в процессе сборки, может использоваться для измерения качества программного проекта. В первую очередь к ней относятся результаты сборки и тестирования, а также показатели качества кода, такие как покрытие тестами. Эта информация может отображаться на участке графика, соответствующем определенному времени, и давать представление о тенденциях качества программного обеспечения.

Другие очень интересные метаданные касаются самого конвейера сборки. Как долго обычно длится сборка? Сколько сборок выполняется за день? Как часто

сборка заканчивается неудачей? Каковы наиболее частые причины сбоев? Сколько времени требуется для исправления сбоя (каково *время восстановления*)? Ответы на эти вопросы дают полезную информацию о качестве процесса непрерывной поставки.

Собранная информация служит хорошим отправным пунктом для дальнейшего совершенствования процесса разработки. Прозрачность непрерывной поставки не только говорит о текущем статусе проекта, но и может привлечь внимание инженеров к определенным проблемным точкам. Главная цель — постоянно совершенствовать программное обеспечение.

Постоянное совершенствование

Вся концепция непрерывной поставки направлена на доставку программного обеспечения с неизменным качеством. Автоматизированные процессы стимулируют использование проверок качества.

Разумеется, высокого качества программного обеспечения нельзя достичь без затрат. Проведение достаточного количество контрольных тестов, а также анализ качества кода требуют определенного времени и усилий. Однако первоначальные затраты на автоматизацию и постоянное повышение качества в конечном счете окупятся и приведут к улучшению программного обеспечения.

Новые функции, а также исправленные ошибки должны быть протестированы во время разработки настолько, чтобы гарантировать, что функциональность работает так, как ожидалось. Автоматизируя тесты и сохраняя их последовательность, разработчики могут быть уверены, что никакие новые ошибки не смогут нарушить функциональность в будущем. То же самое касается и анализа качества кода. Когда анализ настроен, написаны все соответствующие правила, а найденные ошибки устранены, это гарантирует, что никакие новые нарушения в программном обеспечении возникнуть не смогут. При появлении новых ложных сообщений об удачном прохождении проверок правила будут скорректированы, и это предотвратит новые ложные срабатывания в будущем.

Внедрение новых тестовых сценариев, таких как сквозные тесты, поддерживает этот подход. Регрессионные тесты уменьшают риск появления новых ошибок. Ключ к успеху здесь тот же — автоматизация. Как мы увидим в главе 7, вмешательство человека полезно для разработки разумных тестовых сценариев. Однако для обеспечения качества программного обеспечения крайне важно, чтобы эти тесты были автоматизированы и стали частью конвейера. Благодаря этому качество со временем будет улучшаться.

Конечно же, это требует от инженеров присвоить улучшению качества высокий приоритет. Улучшение качества программного обеспечения, а также реструктуризация кода не приносят бизнесу немедленной выгоды. Зато эти усилия окупятся в долгосрочной перспективе за счет того, что можно будет регулярно добавлять новые функции или изменять существующее поведение, будучи уверенными, что при этом ничто другое не будет нарушено.

Резюме

Для продуктивных процессов разработки требуется короткий цикл производства, а также быстрая обратная связь. Автоматизация повторяющихся задач сводит к минимуму время, затрачиваемое на сборку, тестирование и развертывание. Приложения Java EE с нулевыми зависимостями обеспечивают быструю обратную связь, минимизируя время сборки, публикации и развертывания.

Важно определить, какая категория ошибок будет прерывать сборку. Программисты должны знать, что сборка либо прервана из-за обнаруженных ошибок, либо прошла без оповещения об этом. От предупреждений, не влияющих на результат сборки, мало пользы.

Еще один важный момент, на который следует обратить внимание, — это миграция данных. Развертывание приложений без сохранения состояния выполняется сравнительно легко, однако необходимо учитывать схемы базы данных, которые должны соответствовать коду приложения. Развертывание обновлений вместе со сценариями миграции, когда изменения вносятся малыми порциями, позволяет развертывать приложения с нулевым временем простоя. Поэтому приложениям необходимо поддерживать совместимость методом $N - 1$.

Непрерывная поставка зависит от грамотной корпоративной культуры. Недостаточно реализовать только технические потребности — все инженеры-программисты должны придерживаться принципов совместной работы. Потенциальные проблемы сборки, результаты тестов, качество программного обеспечения и состояние развертывания должны быть видны всей команде разработчиков программного обеспечения.

Процессы непрерывной поставки способствуют постоянному совершенствованию программного обеспечения. Добавляемые в конвейер этапы проверки, такие как автоматическое тестирование программного обеспечения, выполняются каждый раз, когда приложение собрано, что позволяет проводить регрессионные тесты и избегать повторения одних и тех же ошибок. Разумеется, это требует от программистов дополнительных усилий по улучшению качества кода. Однако эти усилия, направленные на обеспечение непрерывной поставки, окупаются в долгосрочной перспективе.

Следующая глава посвящена качеству программного обеспечения. В ней будет рассмотрено тестирование корпоративных приложений.

7 Тестирование

Как мы узнали из предыдущей главы, конвейеры непрерывной поставки позволяют программистам поставлять программное обеспечение с постоянной скоростью и стабильным качеством. Для того чтобы гарантировать качество, необходимы автоматические тесты программного обеспечения. Инженеры, разрабатывающие функции, должны быть уверены, что все работает так, как ожидалось. Это особенно важно, когда проект программного обеспечения развивается, изменяется и есть риск нарушить существующее поведение. Программисты должны быть уверены в том, что не возникнет никаких нежелательных побочных эффектов.

В идеале тестов программного обеспечения, содержащихся в конвейере сборки, должно быть достаточно для развертывания приложения в среде эксплуатации (к тому же они не должны требовать дальнейшей ручной проверки).

В этой главе будут рассмотрены следующие темы.

- Требования к программным тестам.
- Различные уровни и области применения тестов.
- Модульные, компонентные, интеграционные, системные тесты и тесты производительности.
- Локальный запуск тестовых сценариев.
- Разработка поддерживаемых тестов.
- Необходимые технологии тестирования.

Необходимость тестирования

Тесты необходимы — они позволяют гарантировать, что определенная функциональность после ввода в эксплуатацию будет вести себя определенным образом. На всех видах производства испытания являются естественной частью процесса. Автомобиль состоит из множества деталей, которые необходимо испытывать независимо друг от друга, а также во взаимодействии. Никто не захочет сесть

за руль машины, если это будет ее первый пробный пробег на реальной улице с живым человеком внутри.

Тесты моделируют поведение продукта при эксплуатации и позволяют проверить компоненты в безопасной среде. Если изготовленные детали ломаются во время тестовых прогонов, это хорошо: это указывает на потенциальные ошибки, потеряны всего лишь время и материалы. Если же детали ломаются в процессе эксплуатации, это может причинить гораздо больший вред.

То же самое можно сказать и о тестах программного обеспечения. Тестовые сбои — это хорошо: в худшем случае потрачены только некоторое время и усилия, в лучшем случае они не позволяют потенциальным ошибкам проявиться в процессе эксплуатации.

Как мы уже знаем, при прохождении тестов вмешательство человека должно быть минимальным. Люди умеют хорошо придумывать тестовые случаи и разрабатывать творческие тестовые сценарии. Однако компьютеры тестируют лучше людей. Проведение сложных тестов — это именно то, в чем компьютеры преуспевают, если им даны четкие инструкции. И чем сложнее со временем становится программное обеспечение, тем больше требуется усилий по проверке его поведения вручную и выше вероятность ошибок. При выполнении повторяющихся задач компьютеры работают лучше и надежнее человека.

Надежные автоматизированные тесты программного обеспечения являются условием быстрых изменений. Автоматические тесты могут проводиться многократно, проверяя все приложение. Сборка запускается много раз в день, при этом каждый раз выполняются все тесты, даже если были внесены незначительные изменения, и к вводу в эксплуатацию допускаются только проверенные версии. В случае тестирования вручную это было бы невозможно.

Автоматизированные тесты повышают надежность процесса непрерывной поставки ПО и обеспечивают уверенность в том, что он протекает штатно. Для непрерывного развертывания, то есть передачи приложения сразу в эксплуатацию, достаточное количество автоматических тестовых сценариев абсолютно необходимо. Когда любая фиксация изменений является потенциальным кандидатом на развертывание в среде эксплуатации, все поведение программного обеспечения должно быть надлежащим образом проверено заранее. Без такой автоматизированной проверки непрерывное развертывание было бы невозможно.

Требования к хорошим тестам

В современном мире программирования все согласны с тем, что тесты имеют решающее значение для получения рабочего программного обеспечения. Но что делает программный тест хорошим? Какие компоненты ПО следует тестировать? И что еще важнее, как разработать хорошие тесты?

В общем случае тесты должны отвечать таким требованиям, как:

- предсказуемость;
- изолированность;

- надежность;
- быстрое выполнение;
- автоматизация;
- удобство сопровождения.

Рассмотрим эти требования подробнее.

Предсказуемость

Прежде всего тесты программного обеспечения должны быть стабильными, предсказуемыми и воспроизводимыми. Одно и то же состояние проекта должно давать одинаковые результаты тестов, то есть они должны либо всегда проходить успешно, либо никогда не выполняться. Тесты, которые заканчиваются то успешно, то неудачно, бесполезны. Они либо отвлекают разработчиков ложными срабатываниями, либо скрывают фактические ошибки, давая ложные сведения об успешном выполнении.

Необходимо учитывать наряду с прочими следующие обстоятельства: текущее время, часовой пояс, локальные языки, случайно генерируемые данные и одновременное выполнение других тестов, которые могут вмешиваться в процесс. Сценарии тестирования следует предсказуемо и явно настраивать так, чтобы эти обстоятельства не влияли на результат. Если же данные факторы влияют на протестированные функциональные возможности, это значит, что нужны дополнительные тестовые сценарии, учитывающие различные конфигурации.

Изолированность

Требование предсказуемости касается и изолированности. Каждый тестовый сценарий должен выполняться по отдельности, не влияя на другие тесты. Изменение и сопровождение тестовых сценариев также не должно влиять на другие тестовые сценарии.

Изолированность тестов влияет не только на предсказуемость и обслуживаемость, но и на воспроизводимость ошибок. Сложные тестовые сценарии способны выполнять множество задач и затрагивают многие моменты, что может затруднить поиск главных причин неудачного прохождения тестов. Однако изолированные тесты с меньшим охватом ограничивают круг возможных проблем и позволяют разработчикам быстрее находить ошибки.

Обычно у корпоративного проекта есть несколько тестовых областей. Как будет показано далее, каждая из них может иметь несколько уровней изоляции тестов. Тесты с небольшим охватом, например модульные, являются сильнее изолированными, чем, например, сквозные тесты. Определенно имеет смысл писать тестовые сценарии для разных областей, что подразумевает различные уровни изоляции тестов.

Надежность

В идеале программные тесты проекта надежно проверяют всю его функциональность. Мантра программиста должна звучать так: программное обеспечение, прошедшее тесты, готово к использованию. Именно к этой цели следует стремиться, постоянно совершенствуя тесты.

Непрерывная поставка кода и особенно непрерывное развертывание требуют надежного и достаточного тестирования. Тесты программного обеспечения являются последним барьером проверки его качества перед развертыванием в среде эксплуатации.

Если надежные тесты успешно пройдены, никаких дополнительных действий не требуется. Поэтому, если их общее выполнение было успешным, они не должны выводить подробные отчеты, так как это становится отвлекающим моментом. Но при неудачных тестах детальное объяснение того, что произошло в процессе тестирования, очень полезно.

Быстрое выполнение

Как уже говорилось, тестирование должно проходить быстро. Быстрые тесты необходимы при разработке конвейеров, обеспечивающих скорую обратную связь. Единственный способ сохранить эффективность конвейера — это сократить время выполнения тестов, особенно если учесть, что с течением времени их количество растет, а сами они постоянно совершенствуются.

Как правило, при выполнении теста основное время затрачивается на запуск технологии тестирования. Особенно много времени уходит на тесты интеграции, в которых используется встроенный контейнер. Время, затрачиваемое на сам тест, в большинстве случаев не так уж и велико.

Продолжительные тесты противоречат идее постоянного повышения качества. Чем больше тестовых случаев и сценариев добавляется в проект, тем дольше тестирование и медленнее обратная связь. В условиях быстро меняющегося мира особенно важно, чтобы тесты программного обеспечения протекали как можно быстрее. В этой главе мы покажем вам, как достичь этой цели, особенно в отношении сквозных тестовых сценариев.

Автоматизация

Автоматизация является обязательным условием обеспечения быстрой обратной связи. Этапы конвейера непрерывной поставки кода должны проходить с минимальным вмешательством человека. Это касается и тестовых сценариев. Программное тестирование и проверка его результатов должны в полном объеме и надежно проходить без взаимодействия с человеком.

Тестовые случаи описывают ожидаемое поведение функциональности и затем выводят результат. Тогда тест будет либо успешно выполняться без дополнительных

уведомлений, либо давать сбой и выдавать подробное объяснение. Тестирование не должно требовать дальнейшего взаимодействия с людьми.

Сценарии с очень большими или сложными тестовыми данными представляют собой особую проблему при автоматизации тестовых случаев. Для того чтобы ее решить, инженеры должны разрабатывать тестовые случаи, которые удобно сопровождать.

Удобство сопровождения

Разработка тестовых случаев — это одно, и совсем другое — поддержка эффективных тестовых случаев с сохранением хорошего покрытия при изменении функциональности. Проблема плохо разработанных тестовых сценариев заключается в том, что, как только функциональность приложения изменится, эти тесты также необходимо будет изменить, что потребует много времени и усилий.

Для разработки тестовых случаев нужны такие же внимание и усилия, как и для написания основного кода программного продукта. Опыт показывает, что тесты, разработанные невнимательным программистом, во многом дублируются и имеют множественные обязанности. Как и основной код приложения, код тестов нуждается в регулярной реструктуризации.

Необходимо иметь возможность изменять и расширять тестовые сценарии, не прилагая особых усилий. В частности, тестовые данные, которые нужно изменить, должны иметь эффективное представление.

Удобное обслуживание тестов является обязательным условием для корпоративных проектов, которые имеют надлежащее тестовое покрытие и при этом достаточно гибки для внесения изменений в их бизнес-логику. Возможность адаптироваться в быстро меняющемся мире требует настраиваемых тестовых сценариев.

Что тестировать

Прежде чем перейти к разговору о том, как создавать эффективные, быстрые, надежные, автоматизированные и удобные в обслуживании тестовые сценарии, рассмотрим, что именно нужно тестировать. Существуют тесты на уровне кода, а также сквозные. Тесты на уровне кода основаны на исходном коде проекта и обычно выполняются в процессе разработки и сборки, тогда как сквозные тесты всех видов работают с уже запущенным приложением.

В зависимости от областей тестирования, которые мы обсудим в следующем разделе, существуют различные уровни тестов: взаимодействующие с классами, несколькими компонентами, корпоративными приложениями или целой средой выполнения. Во всех случаях объект тестирования должен быть изолирован от внешних воздействий. Природа тестов заключается в том, что они проверяют

определенное поведение при определенных условиях. Окружающая среда тестового объекта, такая как тестовые случаи, а также используемые компоненты, должна взаимодействовать с объектом тестирования соответствующим образом. Поэтому тестовый случай управляет тестовым объектом. Сюда входят не только тесты на уровне кода, но и сквозные тесты с имитацией или отчуждением внешних систем.

Главная обязанность тестов программного обеспечения заключается в проверке поведения бизнес-сценариев. В выбранных тестовых случаях должна выполняться определенная логика, которую необходимо проверить до развертывания приложения в среде эксплуатации. Поэтому тесты программного обеспечения должны проверять соответствие приложения требованиям бизнеса. Необходимо охватить также специальные и граничные случаи и отрицательные тесты.

Например, при тестировании функции проверки подлинности требуется убедиться не только в том, что пользователь может войти в систему с правильными учетными данными, но и в том, что он не может войти в систему, используя неверные учетные данные. Граничный тестовый случай для этого примера должен состоять в проверке того, что если срок действия пароля истекает, то после входа пользователя в систему компонент аутентификации сразу отправляет ему уведомление об этом.

Помимо бизнес-поведения, необходимо протестировать технические характеристики и сквозные компоненты. Доступ к базам данных и внешним системам, а также формы коммуникации должны быть проверены на обеих сторонах, чтобы гарантировать работу всей группы. Эти аспекты лучше всего проверять сквозными тестами.

Во всех случаях тестируемый объект не должен изменяться во время теста — он должен функционировать как в процессе производства. Это важно для создания надежных тестов, поведение которых не будет изменяться со временем. Для тестов уровня кода требуется только, чтобы содержимое всех задействованных компонентов не менялось на протяжении тестирования. Для сквозных тестов это касается всего корпоративного приложения, а также установки и настройки среды выполнения приложения.

Определение областей тестирования

Существует несколько сфер и задач тестирования, к которым следует отнестись внимательно. Далее будут перечислены различные области, которые мы подробнее рассмотрим далее в этой главе.

Некоторые названия, такие как «интеграционные тесты», неоднозначно трактуются в разных корпоративных проектах. В этом разделе я приведу согласованные названия областей тестирования, которые будут использоваться в остальной части книги.

Модульные тесты

Модульные тесты предназначены для проверки поведения отдельных модулей приложения. Модульный тест обычно представляет собой одиночный класс, в некоторых случаях — несколько взаимозависимых классов.

Работают такие тесты на уровне кода. Обычно они выполняются в среде IDE в процессе разработки, а также в ходе сборки до того, как приложение будет упаковано. Модульные тесты протекают быстрее прочих. Они реализуют ограниченный набор функций, которые можно легко создать на уровне кода. Потенциальные зависимости между модулями моделируются с помощью макетов или фиктивных классов.

Компонентные тесты

Компонентные тесты проверяют поведение сопряженных компонентов. Они охватывают более одного модуля, но тоже работают на уровне кода. Компонентные тесты нацелены на объединение нескольких компонентов и проверяют взаимозависимое поведение, не настраивая контейнерные среды.

Целью компонентных тестов является обеспечение большей степени интеграции, чем с помощью модульных тестов, без запуска приложения в потенциально медленных имитируемых средах. Подобно модульным тестам, компонентные используют моделирование функциональности для разграничения и имитирования ограничений теста. Встроенный или удаленный контейнер предприятия для этого тестирования не требуется.

Интеграционные тесты

Существуют разногласия по поводу того, что такое интеграционные тесты и как их следует разрабатывать. Целенаправленная интеграция может происходить на разных уровнях.

Я буду использовать этот термин, поскольку он широко распространен в экосистеме Java и соглашениях Maven. Интеграционные тесты выполняются на уровне кода, обеспечивая интеграцию нескольких модулей и компонентов, и обычно реализуют несколько более или менее сложных схем тестирования. В этом основное отличие интеграционных тестов от компонентных.

Интеграционные тесты имеют ту же область действия, что и компонентные, поскольку объединяют несколько модулей, однако в данном случае основное внимание уделяется интеграции. Имеется в виду скорее технологическая интеграция, чем интеграция бизнес-логики. Например, управляемые компоненты могут использовать CDI-внедрение для получения определенных зависимостей с применением квалификаторов или CDI-генераторов. Программистам необхо-

димо проверить, правильно ли были проделаны все *вспомогательные операции* по CDI, то есть были ли использованы правильные аннотации, без развертывания приложения на сервере.

Структуры тестирования запускают встроенную среду выполнения, которая собирает несколько компонентов и запускает для них тесты на уровне кода.

Компонентные тесты, напротив, сосредоточены исключительно на бизнес-логике и ограничены простыми зависимостями, которые легко распознаются без применения сложных контейнеров. В целом компонентные тесты предпочтительнее для тестирования бизнес-приложений, поскольку содержат меньше изменяемых частей и, как правило, работают быстрее.

Системные тесты

Термин «системные тесты» тоже иногда имеет разный смысл. В данном контексте мы будем так называть тестовые случаи, требующие запуска всего приложения или системы в целом и проверяющие сценарии использования от начала до конца. Иногда их называют также приемочными или интеграционными. Однако в этой книге для обозначения сквозных тестов будем последовательно применять термин «системные тесты».

Системные тесты очень важны для проверки того, что развернутое приложение, включая как бизнес-логику, так и технические компоненты, работает должным образом. С учетом того, что бóльшая часть бизнес-логики уже проверена модульными и компонентными тестами, системные тесты позволяют убедиться в правильности общего поведения приложения, включая все внешние системы. Сюда входит также проверка того, что все функции интегрированы верно и правильно взаимодействуют в системной среде.

Чтобы приложение представляло ценность для бизнеса, недостаточно просто включить в него бизнес-логику — нужны способы доступа к ней. Это также необходимо проверить от начала и до конца.

Поскольку эта книга посвящена серверным приложениям, тесты на уровне пользовательского интерфейса в ней не рассматриваются. Такие тесты включают в себя комплексные тесты пользовательского интерфейса, а также тесты на реактивность UI. Обычно программисты разрабатывают тесты пользовательского интерфейса с использованием таких тестовых технологий, как *Arquillian Graphene*. Методы системного тестирования, описанные в этой главе, также применимы к тестам на уровне UI.

Тесты производительности

Тесты производительности предназначены для того, чтобы проверить нефункциональный показатель: как система работает с точки зрения скорости реагирования и правильного поведения при определенных нагрузках.

Необходимо обеспечить, чтобы приложение имело ценность для бизнеса не только в лабораторных условиях, но и при эксплуатации. В реальных условиях нагрузка на систему может сильно отличаться от таковой в условиях разработки. Это зависит от характера приложения и вариантов его использования. Общедоступные приложения также рискуют стать объектом атак типа «отказ в обслуживании».

Тесты производительности — полезный инструмент для выявления потенциальных проблем с производительностью приложения. К ним относятся, например, тесты на утечку ресурсов, неправильную конфигурацию, тупиковые ситуации и отсутствие задержек. Исследование поведения приложения с имитацией рабочей нагрузки позволяет выявить подобные проблемы.

Однако, как мы увидим в главе 9, тесты производительности не всегда дают возможность спрогнозировать производительность в условиях эксплуатации или способы настройки приложения, позволяющие достичь должной производительности. Они должны использоваться лишь как барьер против очевидных ошибок, обеспечивая быструю обратную связь.

Далее в этой книге я буду применять термин «тесты производительности» для описания тестов производительности, а также нагрузочных и стресс-тестов, которые ставят приложение в условия высокой нагрузки.

Стресс-тесты

Как и тесты производительности, стресс-тесты направлены на то, чтобы создать для системы стрессовые условия и проверить, правильно ли она себя ведет в ненормальных ситуациях. В то время как тесты производительности в основном нацелены на определение производительности приложения с точки зрения правильного выполнения функций и стабильности, стресс-тесты могут охватывать все особенности поведения приложения, в том числе попытки вывести систему из строя. К последним относятся недопустимые вызовы, несоблюдение условий обмена данными и случайные, непредусмотренные события из среды выполнения. Это далеко не полный список тестов — его подробное рассмотрение выходит за рамки книги.

Однако приведу несколько примеров. Так, стресс-тест может проверять приложение на предмет неправильного использования HTTP-соединений, таких как лавинная адресация SYN, различные DDoS-атаки, неожиданные отключения инфраструктуры, а также так называемое случайное тестирование или тесты на «защиту от дурака».

Создание сложного пакета тестов, включающего в себя большое количество стресс-тестов, выходит за рамки большинства проектов. Однако для корпоративных проектов имеет смысл провести несколько разумных стресс-тестов, соответствующих используемой среде.

Реализация тестирования

Изучив причины, требования и разные области тестирования, более подробно рассмотрим, как создавать тестовые случаи в проектах Java Enterprise.

Модульные тесты

Модульные тесты проверяют поведение отдельных модулей приложения. В Java EE это обычно относится к отдельным сущностям, контурам и управляющим классам.

Для модульного тестирования отдельного класса не требуется исчерпывающих тестовых случаев. В идеале достаточно создать экземпляр тестового объекта и настроить минимальные зависимости, чтобы иметь возможность вызывать и проверять его бизнес-функциональность.

Современная среда Java EE поддерживает этот подход. Компоненты Java EE, такие как EJB, а также управляемые компоненты CDI можно легко проверить, просто создав экземпляры классов. Как мы уже знаем, современные корпоративные компоненты представляют собой простые Java-объекты, включающие аннотации, без расширения или реализации технически обоснованных суперклассов и интерфейсов, так называемые представления без интерфейса.

Это позволяет тестировать экземпляры классов EJB и CDI, подключая их по мере необходимости. Используемые делегаты, такие как внедренные элементы управления, не имеющие отношения к тестовому сценарию, заменяются макетами. Поступая таким образом, мы определяем границы тестового случая: что нужно протестировать, а что не имеет значения. Макетирование делегатов позволяет проверить взаимодействие с тестовым объектом.

Макет объекта имитирует поведение настоящего экземпляра данного типа. Методы вызова макетов обычно возвращают только фиктивные, или макетные, значения. Тестируемые объекты не знают, что обмениваются данными с макетным объектом. Поведение макетов, а также проверка вызванных методов определяются тестовым сценарием.

Реализация

Начнем с модульного теста для основного компонента Java EE. Контур `CarManufacturer` реализует определенную бизнес-логику и вызывает управляющий элемент-делегат CarFactory:

```
@Stateless
public class CarManufacturer {
    @Inject
    CarFactory carFactory;

    @PersistenceContext
```

```
    EntityManager entityManager;

    public Car manufactureCar(Specification spec) {
        Car car = carFactory.createCar(spec);
        entityManager.merge(car);
        return car;
    }
}
```

Поскольку контур EJB является простым Java-классом, его можно создать и настроить в модульном тесте. Наиболее популярной технологией тестирования Java-модулей является *JUnit* в сочетании с *Mockito* для создания макетов. В следующем фрагменте кода показан тестовый случай для класса `CarManufacturer`, в котором создается экземпляр объект для граничного теста и задействуется Mockito для макетирования делегатов:

```
import org.junit.Before;
import org.junit.Test;
import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.*;

public class CarManufacturerTest {

    private CarManufacturer testObject;

    @Before
    public void setUp() {
        testObject = new CarManufacturer();
        testObject.carFactory = mock(CarFactory.class);
        testObject.entityManager = mock(EntityManager.class);
    }

    @Test
    public void test() {
        Specification spec = …
        Car car = …

        when(testObject.entityManager.merge(any())).then(a->
            a.getArgument(0));
        when(testObject.carFactory.createCar(any())).thenReturn(car);

        assertThat(testObject.manufactureCar(spec)).isEqualTo(car);

        verify(testObject.carFactory).createCar(spec);
        verify(testObject.entityManager).merge(car);
    }
}
```

При выполнении теста фреймворк JUnit однократно запускает тестовый класс `CarManufacturerTest`.

Метод `@Before`, в данном случае `setUp()`, выполняется каждый раз перед запуском метода `@Test`. Аналогично методы с аннотацией `@After` запускаются после

каждого выполнения теста. Однако методы `@BeforeClass` и `@AfterClass` реализуются только один раз для каждого тестового класса — до и после выполнения соответственно.

Методы Mockito, такие как `mock()`, `when()` и `verify()`, используются для создания, настройки и проверки макетируемого поведения соответственно. Макетные объекты настроены на определенное поведение. После выполнения теста они могут проверить, были ли вызваны ими определенные функции.

Нельзя не признать, что это очень простой пример. Но в нем представлена суть модульного тестирования основных компонентов. Для проверки поведения контура не требуется других специальных исполнителей тестов или встроенных контейнеров. В отличие от специальных исполнителей тестов фреймворк JUnit позволяет запускать модульные тесты с очень высокой скоростью. Современное оборудование позволяет выполнять сотни таких примеров почти мгновенно. Их время запуска короткое, а остальное — это просто выполнение Java-кода с небольшими накладными расходами на фреймворк тестирования.

Некоторые читатели, возможно, заметили, что область видимости класса `CarManufacturer` ограничена пределами пакета. Это сделано для того, чтобы обеспечить лучшую тестируемость и иметь возможность назначать делегат для экземпляров классов. Тестовые классы, которые находятся в том же пакете, что и контур, могут изменять свои зависимости. Однако инженеры могут возразить, что это нарушает инкапсуляцию контура. Теоретически они правы, но ни один вызывающий объект не сможет изменить ссылки после того, как компоненты начнут выполняться в корпоративном контейнере. Объект, на который указывает ссылка, в действительности является не делегатом, а его посредником, поэтому реализация CDI может предотвратить неправильное использование. Конечно, можно внедрить объект-макет, задействуя отражение или внедрение на основе конструктора. Однако внедрение на основе поля в сочетании с непосредственной настройкой зависимостей для тестовых случаев обеспечивают лучшую читаемость кода при том же поведении приложения. В настоящее время при разработке многих корпоративных проектов применяется внедрение зависимостей на основе поля с видимостью в пределах пакета.

Другой вопрос заключается в том, что лучше использовать: нестандартные исполнители тестов JUnit, такие как `MockitoJUnitRunner`, в сочетании со специальными макетными аннотациями или же простую настройку, как показано ранее. В следующем фрагменте кода представлен более подробный пример с нестандартным исполнителем тестов:

```
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class CarManufacturerTest {

    @InjectMocks
```

```
    private CarManufacturer testObject;

    @Mock
    private CarFactory carFactory;

    @Mock
    private EntityManager entityManager;

    @Test
    public void test() {
        …
        when(carFactory.createCar(any())).thenReturn(car);
        …
        verify(carFactory).createCar(spec);
    }
}
```

Нестандартный исполнитель тестов Mockito позволяет программистам настраивать тесты с меньшим количеством кода, а также создавать внедрения с видимостью в пределах класса обслуживания. Применение простого подхода, как было показано ранее, обеспечивает большую гибкость для сложных макетных сценариев. Впрочем, то, какой именно метод использовать для запуска и определения макетов Mockito, — в сущности, дело вкуса.

В JUnit есть дополнительная функциональность — параметризованные тесты. Это позволяет создавать однообразные тестовые сценарии с неодинаковыми входными и выходными данными. Метод `manufactureCar()` можно протестировать с разными входными данными, что дает чуть различающиеся выходные результаты. Параметризованные тестовые случаи позволяют более эффективно разрабатывать эти сценарии. В следующем примере показан код для таких тестовых случаев:

```
import org.junit.runners.Parameterized;

@RunWith(Parameterized.class)
public class CarManufacturerMassTest {

    private CarManufacturer testObject;

    @Parameterized.Parameter(0)
    public Color chassisColor;

    @Parameterized.Parameter(1)
    public EngineType engineType;

    @Before
    public void setUp() {
        estObject = new CarManufacturer();
        testObject.carFactory = mock(CarFactory.class);
        …
    }
    @Test
    public void test() {
```

```
        // chassisColor & engineType
        …
    }

    @Parameterized.Parameters(name = "chassis: {0}, engine type: {1}")
    public static Collection<Object[]> testData() {
        return Arrays.asList(
                new Object[]{Color.RED, EngineType.DIESEL, ...},
                new Object[]{Color.BLACK, EngineType.DIESEL, …}
        );
    }
}
```

Параметризованные тестовые классы создаются и выполняются после данных в методе тестовых данных `@Parameters`. Каждый элемент в возвращаемой коллекции приводит к тому, что тест проводится отдельно. Тестовый класс присваивает значения свойствам, и тест продолжает выполняться как обычно. В тестовых данных содержатся входные параметры тестов, а также ожидаемые значения.

Преимущество параметризованного подхода заключается в том, что он позволяет программистам создавать новые тестовые случаи, просто добавляя строки в метод `testData()`. В предыдущем примере показано применение параметризованного модульного теста в сочетании с макетами. Такая комбинация возможна только при использовании простого подхода Mockito, как было описано ранее, но не с `MockitoJUnitRunner`.

Технология

В рассмотренных примерах был применен фреймворк JUnit версии 4, которая на момент написания этой книги была распространена наиболее широко. Mockito задействуется для макетирования объектов и обеспечивает достаточную для большинства сценариев гибкость. Для того чтобы подтверждать условия, в примерах была использована библиотека сопоставления тестов *AssertJ*. Она предоставляет функциональные возможности для проверки состояния объектов с применением эффективных вызовов цепочек методов.

Эти технологии являются примерами инструментов, необходимых для тестирования. Однако здесь мы не намерены настаивать на использовании тех или иных функций, а лишь хотим продемонстрировать конкретные разумные варианты соблюдения условий тестирования. Можно применять и другие технологии, обеспечивающие аналогичные функциональность и преимущества.

Типичным примером является широко распространенная в качестве библиотеки сопоставления тестов технология анализаторов Hamcrest или реже используемый фреймворк для модульного тестирования *TestNG*.

К тому моменту, когда вы это прочитаете, появится версия JUnit 5, предоставляющая некоторые дополнительные функции, особенно для динамических тестов. Подобно параметризованным тестам, динамические тесты применяются потому, что допускают программное и динамическое определение тестовых случаев.

Компонентные тесты

Компонентные тесты проверяют поведение сопряженных компонентов. Они обеспечивают бóльшую интеграцию, чем модульные тесты, не требуя запуска приложения в имитируемых средах выполнения.

Причины использования

Поведение сопряженных функций, представленных несколькими взаимозависимыми классами, необходимо проверить, чтобы протестировать их взаимодействие. Компонентные тесты должны выполняться так же быстро, как и модульные, все еще изолируя функции, то есть тестируя сопряженные модули. Поэтому эти тесты нацелены на обеспечение быстрой обратной связи путем интеграции большего количества логики, чем в обычных модульных тестах. Компонентные тесты проверяют работу бизнес-сценариев от контура до задействованных элементов управления.

Компонентные тесты на уровне кода возможны, поскольку подавляющее большинство управляемых компонентов используют довольно простые делегаты. Внедренные типы в большинстве случаев могут быть распознаны непосредственно, без интерфейсов или квалификаторов, и созданы и внедрены практически без встроенных контейнеров. Это позволяет реализовать компонентные тесты с теми же фреймворками тестирования, что и для модульных тестов. Все необходимые делегаты и макеты настраиваются как часть тестового случая. Сценарий тестирования, который мы хотим показать, проверяет все — от начала бизнес-сценария до внедренных элементов управления.

В следующих примерах рассмотрим, как реализовать компонентные тесты с помощью некоторых базовых методов контроля качества кода, которые помогают создавать обслуживаемые тесты.

Реализация

Предположим, что нужно протестировать весь бизнес-сценарий приложения по производству автомобилей, показанный в предыдущем примере в подразделе «Модульные тесты». Для создания автомобиля применяется делегат `CarFactory`, а затем результат сохраняется в базе данных. Тестирование на уровне базы данных выходит за рамки этого теста, поэтому вместо диспетчера сущностей будет использован макет.

В следующем фрагменте кода показан компонентный тест бизнес-сценария производства автомобиля:

```
public class ManufactureCarTest {

    private CarManufacturer carManufacturer;

    @Before
    public void setUp() {
        carManufacturer = new CarManufacturer();
```

```
        carManufacturer.carFactory = new CarFactory();
        carManufacturer.entityManager = mock(EntityManager.class);
    }

    @Test
    public void test() {
        when(carManufacturer.entityManager.merge(any())).then(a->
            a.getArgument(0));

        Specification spec = …
        Car expected = …

    assertThat(carManufacturer.manufactureCar(spec)).isEqualTo(expected);
        verify(carManufacturer.entityManager).merge(any(Car.class));
    }
}
```

Этот пример очень похож на предыдущий, за исключением того, что здесь для создания `CarFactory` задействована реальная бизнес-логика. Макеты, представляющие границы тестового случая, проверяют правильность поведения.

Данный метод пригоден для простых сценариев использования, однако для более сложных реальных сценариев применять его несколько наивно. Границы тестового случая показаны в тестовом классе, поскольку делегат `CarFactory` является самодостаточным и не внедряет дополнительные элементы управления. Конечно, все взаимозависимые модули, которые являются частью компонентного теста, могут определять делегатов. В зависимости от характера теста и сценария использования эти вложенные делегаты также должны быть созданы или смакетированы.

В итоге для настройки тестового примера требуются большие усилия. Мы могли бы применить функциональные возможности тестового фреймворка, такие как аннотации Mockito. Таким образом в тестовый пример внедрялись бы все классы, участвующие в тестовом случае. Программисты указывают, какие из них должны создаваться, а какие — заменяться макетами. Mockito предоставляет функциональные возможности для распознавания ссылок, достаточные для большинства сценариев использования.

В следующем фрагменте кода представлен компонентный тест аналогичного сценария, на этот раз с делегатом `CarFactory`, с вложенными зависимостями `AssemblyLine` и `Automation`. Они макетируются в тестовом случае:

```
@RunWith(MockitoJUnitRunner.class)
public class ManufactureCarTest {

    @InjectMocks
    private CarManufacturer carManufacturer;

    @InjectMocks
    private CarFactory carFactory;

    @Mock
    private EntityManager entityManager;

    @Mock
```

```
    private AssemblyLine assemblyLine;

    @Mock
    private Automation automation;

    @Before
    public void setUp() {
        carManufacturer.carFactory = carFactory;

        // настройка требует макетируемого поведения, например:
        when(assemblyLine.assemble()).thenReturn(...);
    }

    @Test
    public void test() {
        Specification spec = …
        Car expected = …

    assertThat(carManufacturer.manufactureCar(spec)).isEqualTo(expected);
        verify(carManufacturer.entityManager).merge(any(Car.class));
    }
}
```

Функция Mockito `@InjectMocks` пытается распознать ссылки на объекты с макетными объектами, внедренными в тестовый случай как `@Mock`. Ссылки задаются с использованием отражения. Если контуры или элементы управления создают новые делегаты, то они должны быть определены в тестовых случаях по крайней мере как объект `@Mock` во избежание исключения `NullPointerException`. Однако этот метод лишь частично улучшает ситуацию, поскольку приводит к тому, что в тестовом классе определяется много зависимостей.

Если придерживаться этого подхода, то вскоре корпоративный проект по мере роста числа компонентных тестов становится слишком детализированным, с большим объемом дублирования.

Для того чтобы сделать тестовый код не столь громоздким и сократить дублирование, можно было бы ввести тестовый суперкласс для каждого бизнес-сценария. Этот суперкласс содержал бы все определения `@Mock` и `@InjectMock`, устанавливал необходимые зависимости, создавал делегаты и макеты. Однако такие тестовые суперклассы содержат много неявной логики, которая определяется делегатами и используется в расширенных тестовых случаях. Из-за этого появляются тестовые случаи, тесно связанные с обычными суперклассами, что в итоге приводит к неявному связыванию тестовых случаев.

Делегирование тестовых компонентов

Вместо расширения лучше использовать делегирование.

Логика макетирования и проверки, зависящая от применяемых компонентов, делегируется специальным тестовым объектам. Таким образом, делегаты инкапсулируют эту логику и управляют ею по отдельности.

В следующем фрагменте кода показан тестовый случай с использованием компонентов, определяющих зависимости для `carManufacturer` и `carFactory`:

```
public class ManufactureCarTest {

    private CarManufacturerComponent carManufacturer;
    private CarFactoryComponent carFactory;

    @Before
    public void setUp() {
        carFactory = new CarFactoryComponent();
        carManufacturer = new CarManufacturerComponent(carFactory);
    }

    @Test
    public void test() {
        Specification spec = …
        Car expected = …

    assertThat(carManufacturer.manufactureCar(spec)).isEqualTo(expected);

        carManufacturer.verifyManufacture(expected);
        carFactory.verifyCarCreation(spec);
    }
}
```

Тестовые зависимости `Component` описывают объявленные зависимости и макеты, включая настройку и проверку поведения для наших тестовых случаев. Смысл в том, чтобы определить компоненты, которые можно повторно использовать в рамках многокомпонентных тестов, привязывая к ним аналогичную логику.

В следующем фрагменте кода показано определение `CarManufacturerComponent`:

```
public class CarManufacturerComponent extends CarManufacturer {

    public CarManufacturerComponent(CarFactoryComponent carFactoryComponent) {
        entityManager = mock(EntityManager.class);
        carFactory = carFactoryComponent;
    }

    public void verifyManufacture(Car car) {
        verify(entityManager).merge(car);
    }
}
```

Класс находится в том же пакете, что и `CarManufacturer`, но в исходных кодах тестов. Это может быть подкласс контура для добавления макетов и логики проверки. В данном примере он зависит от компонента `CarFactory`, который также обеспечивает дополнительную тестовую логику:

```
public class CarFactoryComponent extends CarFactory {

    public CarFactoryComponent() {
```

```
        automation = mock(Automation.class);
        assemblyLine = mock(AssemblyLine.class);
        when(automation.isAutomated()).thenReturn(true);
    }

    public void verifyCarCreation(Specification spec) {
        verify(assemblyLine).assemble(spec);
        verify(automation).isAutomated();
    }
}
```

Эти компоненты являются тестовыми объектами многократного применения, которые устанавливают определенные зависимости и настраивают поведение макетов. Их можно повторно задействовать в нескольких компонентных тестах и расширить без ущерба для использования.

Цель этих примеров — дать вам представление о возможностях обслуживаемых тестов. Для повторного применения компонентов необходимо учитывать дополнительные методы реструктуризации в различных ситуациях, например с использованием конфигурации, подобной шаблону «Строитель». Подробнее о том, как писать код обслуживаемых тестов, читайте в разделе «Обслуживание тестовых данных и сценариев».

Преимущество компонентных тестов заключается в том, что они работают так же быстро, как и модульные, и при этом проверяют более сложную интеграционную логику. Сложная логика реализуется путем делегирования и инкапсуляции, что повышает удобство обслуживания. Код и накладные расходы, неизбежные при настройке, невелики.

С помощью компонентных тестов имеет смысл проверять сопряженную бизнес-логику. Вызовы бизнес-сценариев проверяются на бизнес-уровне, а низкоуровневые технические детали макетируются.

Технология

В этих примерах снова продемонстрированы простые методы тестирования с помощью JUnit и Mockito. При использовании этих методов проверки качества кода можно писать удобно обслуживаемые подробные тестовые случаи с ограниченными затратами на конфигурацию.

Компонентные тесты, реализованные так, как показано в этих примерах, представляют собой практический подход к подключению компонентов, использующих непосредственное внедрение зависимостей. Если в готовом коде задействуются CDI-генераторы и квалификаторы, то логика внедрения тестовых компонентов изменится соответствующим образом.

Целью компонентных тестов является проверка поведения сопряженных модулей в бизнес-сценариях. Обычно они не проверяют связанные технические компоненты. Для того чтобы проверить правильность применения внедрения CDI, например, с точки зрения специальных классификаторов, генераторов и областей видимости, рекомендуется использовать интеграционные тесты.

Однако существуют тестовые технологии, которые обеспечивают внедрение зависимостей в тестовых случаях, такие как *CDI-Unit* или более сложный фреймворк тестирования *Aquillian*. Тестовые случаи с использованием этих фреймворков выполняются в контейнерах, не только встроенных, но даже удаленных, и могут дополнительно проверять интеграцию компонентов.

Сложные тестовые фреймворки, безусловно, позволяют создавать тестовые случаи, более близкие к реальному корпоративному приложению, но их проблемой является медленный запуск приложений. Контейнеры обычно запускаются и настраиваются для каждого тестового случая в отдельности, что обычно занимает несколько сотен миллисекунд или еще больше. На первый взгляд это не так уж много, но по мере увеличения количества тестов затраты времени становятся заметными.

Поэтому для компонентных тестов, целью которых является только проверка бизнес-сценариев, предпочтительны более быстрые и легкие методы, такие как представленный ранее. Благодаря тому что компонентные и модульные тесты быстры по своей природе, они выполняются по умолчанию во время сборки проекта. Они должны быть стандартным способом проверки бизнес-логики приложения.

В следующем примере приведены интеграционные тесты на уровне кода, в которых используются имитируемые контейнеры.

Интеграционные тесты

При компонентном тестировании сопряженная бизнес-логика проверяется путем выполнения изолированных и быстрых тестов. Однако эти тесты не охватывают сложное поведение интеграции Java EE, такое как внедрения, специальные квалификаторы, CDI-события или области действия.

Целью интеграционных тестов является проверка технического сопряжения компонентов в рамках корпоративной системы. Это касается таких проблем, как конфигурация основных компонентов Java EE, связь с внешними системами и хранение данных. Правильно ли аннотированы компоненты Java EE? Преобразует ли JSON-B данные в нужный формат JSON? Правильно ли описано преобразование JPM ORM?

Суть интеграционных тестов на уровне кода заключается в обеспечении более быстрой обратной связи путем проверки интеграции без создания и развертывания приложения в тестовой среде.

Встроенные контейнеры

Поскольку технология модульных тестов не учитывает особенности Java EE, интеграционные тесты требуют более сложных тестовых функций в виде контейнеров. Есть несколько технологий, которые запускают встроенный контейнер и делают доступными части приложения.

Примером такой технологии является CDI-Unit. Она предоставляет функциональные возможности для запуска тестовых случаев в CDI-контейнере,

что позволяет программистам улучшать и изменять конфигурацию. CDI-Unit сканирует зависимости тестируемых объектов и конфигурирует их соответствующим образом. Все необходимые макеты и специфическое тестовое поведение определяются в виде деклараций. Управляемый компонент, такой как контур в приложении для производства автомобилей, настраивается в тестовом случае со всеми необходимыми зависимостями и макетами.

Этот метод обнаруживает ошибки конфигурации, такие как отсутствие CDI-аннотаций. В следующем фрагменте кода показан тест приложения для производства автомобилей, аналогичный показанному ранее компонентному тесту, который создает контур:

```
import org.jglue.cdiunit.CdiRunner;

@RunWith(CdiRunner.class)
public class ManufactureCarIT {

    @Inject
    CarManufacturer carManufacturer;

    @Mock
    EntityManager entityManager;

    @Before
    public void setUp() {
        carManufacturer.entityManager = entityManager;
    }

    @Test
    public void test() {
        Specification spec = …
        Car expected = …

    assertThat(carManufacturer.manufactureCar(spec)).isEqualTo(expected);
        verify(entityManager).merge(expected);
    }
}
```

Специальный исполнитель тестов JUnit находит исполняемые компоненты, внедряемые в тестовых случаях, и распознает их. Поскольку CDI-Unit поддерживает только стандарт CDI, а не полный API Java EE, тест явно макетирует и настраивает менеджер сущностей. Все остальные элементы управления, такие как фабрика, автоматизация и сборочная линия автомобилей, создаются и вводятся соответствующим образом.

Тесты CDI-Unit могут быть расширены, чтобы обслуживать более сложные сценарии. Можно генерировать компоненты, которые используются в пределах области тестирования.

Однако у этой технологии, безусловно, есть свои ограничения. CDI-Unit удобна, когда надо быстро проверить конфигурацию и взаимодействие управляемых компонентов.

Еще одна, более сложная технология тестирования приложений называется Arquillian. Она объединяет интеграционные тестовые случаи в развертываемые архивы, управляет корпоративными контейнерами, как встроенными, так и удаленными, и развертывает, выполняет и проверяет тестовые архивы. Это позволяет расширять тестовые сценарии с тестовым поведением, зависящим от сценария.

Преимуществом Arquillian является работа с контейнерами с полной поддержкой Java EE. Это позволяет проводить интеграционные тесты в сценариях, приближенных к условиям эксплуатации.

В следующем фрагменте кода представлен простой пример развертывания контура приложения по производству автомобилей во встроенном корпоративном контейнере под управлением Arquillian:

```
import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.asset.EmptyAsset;
import org.jboss.shrinkwrap.api.spec.WebArchive;

@RunWith(Arquillian.class)
public class ManufactureCarIT {

    @Inject
    CarManufacturer carManufacturer;

    @Deployment
    public static WebArchive createDeployment() {
        return ShrinkWrap.create(WebArchive.class)
                .addClasses(CarManufacturer.class)
                // ... добавить остальные нужные зависимости
                .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml");
    }

    @Test
    public void test() {
        Specification spec = …
        Car expected = …

    assertThat(carManufacturer.manufactureCar(spec)).isEqualTo(expected);
    }
}
```

В этом тестовом случае создается динамический веб-архив, который доставляет контур приложения и требуемых делегатов и развертывает их во встроенном контейнере. Сам тест может внедрять и вызывать методы для конкретных компонентов.

Контейнер не обязательно должен запускаться как встроенный, он может быть управляемым или удаленным. Если контейнер работает дольше, чем требуется для теста, то можно сократить время за счет запуска контейнера и выполнять тесты намного быстрее.

Реализация этих интеграционных тестов занимает довольно много времени, но зато точнее имитирует реальное поведение в среде эксплуатации. Ошибки

в управляемых компонентах будут обнаружены в процессе разработки, перед отправкой приложения. Гибкость и настраиваемость Arquillian путем включения дополнительных определений компонентов, которые находятся в области тестирования, позволяют использовать удобные тестовые сценарии.

Однако в этом примере лишь слегка затронута функциональность встроенных контейнерных тестов. Тестовые фреймворки, такие как Arquillian, могут применяться для проверки интеграции конфигурации контейнера, коммуникации, хранения данных и пользовательского интерфейса. В оставшейся части этой главы мы рассмотрим некоторые недостатки интеграционных тестов, работающих с имитируемой или встроенной средой.

Встроенные базы данных

Преобразование при сохранении объектов предметной области обычно определяется с помощью аннотаций JPA. Проверка этого преобразования перед фактическим развертыванием сервера позволяет избежать случайных ошибок и сэкономить время.

Для того чтобы проверить правильность преобразования при сохранении в базе данных, необходима сама база данных. Помимо использования развернутых экземпляров базы данных в среде выполнения, встроенные базы данных обеспечивают аналогичную проверку с быстрой обратной связью. Для доступа к этой функциональности можно применить встроенные контейнерные тесты, работающие в таких фреймворках, как Arquillian. Однако для простейшей проверки нет необходимости запускать приложение внутри контейнера.

JPA поставляется с возможностью автономного запуска в любой среде Java SE. Используя ее, можно писать тестовые случаи, которые принимают конфигурацию JPA и соединяются со встроенной или локальной базой данных.

Предположим, что в процессе производства автомобилей изготавливается и собирается некая деталь автомобиля. Эта сущность как объект предметной области отображается в JPA следующим образом:

```
@Entity
@Table(name = "car_parts")
public class CarPart {

    @Id
    @GeneratedValue
    private long id;

    @Basic(optional = false)
    private String order;

    @Enumerated(STRING)
    @Basic(optional = false)
    private PartType type;

    …
}
```

Для того чтобы проверить правильность сохранения данных, компонент тестовой сущности должен по крайней мере сохраняться в базе данных и загружаться оттуда. В следующем фрагменте кода показан интеграционный тест, который устанавливает автономное JPA-сохранение:

```
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class CarPartIT {

    private EntityManager entityManager;
    private EntityTransaction transaction;

    @Before
    public void setUp() {
        entityManager =
            Persistence.createEntityManagerFactory("it").createEntityManager();
        transaction = entityManager.getTransaction();
    }

    @Test
    public void test() {
        transaction.begin();

        CarPart part = new CarPart();
        part.setOrder("123");
        part.setType(PartType.CHASSIS);
        entityManager.merge(part);

        transaction.commit();
    }
}
```

Поскольку сохранение протекает автономно, то нет контейнера, который бы занимался обработкой транзакций. В тестовом случае это выполняется программно, так же как и настройка менеджера сущностей, с помощью блока управления данными `it`. Блок управления данными конфигурируется в области тестирования `persistence.xml`. Для этой цели достаточно определить конфигурацию локального транзакционного модуля ресурса:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">

    <persistence-unit name="it" transaction-type="RESOURCE_LOCAL">
        <class>com.example.cars.entity.CarPart</class>

        <exclude-unlisted-classes>true</exclude-unlisted-classes>
        <properties>
            <property name="javax.persistence.jdbc.url"
                      value="jdbc:derby:./it;create=true"/>
```

```
            <property name="javax.persistence.jdbc.driver"
                      value="org.apache.derby.jdbc.EmbeddedDriver"/>
            <property name="javax.persistence.schema-generation.database.action"
                      value="drop-and-create"/>
        </properties>
    </persistence-unit>
</persistence>
```

Задействованные классы сущностей, такие как `CarPart`, должны быть указаны явно, поскольку нет контейнера, который занимался бы обработкой аннотаций. Конфигурация JDBC указывает на встроенную базу данных, в этом случае *Apache Derby*.

В корпоративный проект входит не вся реализация Java EE — только API. Таким образом, реализация JPA, такая как *EclipseLink*, а также база данных Derby добавляются как тестовые зависимости.

Этот интеграционный тест обеспечивает более быструю обратную связь, информирующую о несоответствиях конфигурации и случайных ошибках, локально проверяя преобразование при сохранении данных. Приведенный тестовый случай завершится ошибкой, потому что свойство `order` для типа `CarPart` не может быть преобразовано и сохранено, так как `order` является зарезервированным ключевым словом SQL. Для того чтобы устранить эту ошибку, надо изменить преобразование столбцов, например переименовав столбец: `@Column (name = "part_order")`.

Это типичный пример ошибки, которая может быть допущена программистами при конфигурировании сохранения данных. Предотвращая такие ошибки, которые в противном случае не будут обнаружены до момента развертывания, мы обеспечиваем более быструю обратную связь, экономим время и усилия.

Разумеется, таким образом обнаруживаются не все несоответствия интеграции и базы данных. Поскольку контейнер не используется, ошибки сохранения, например связанные с параллельными транзакциями, не будут выявлены, пока не будут выполнены полноценные системные тесты. Тем не менее такое тестирование является полезной первичной проверкой в конвейере.

Запуск интеграционных тестов

Внимательные читатели, вероятно, заметили, что в интеграционных тестах существует соглашение об именах: они заканчиваются на `IT`, что означает integration test. Такое именование основано на соглашении об именах Maven, за исключением тестовых классов, соответствующих шаблону именования `Test`, на этапе тестирования. Классами с именами, заканчивающимися на `IT`, будет управлять другой модуль жизненного цикла.

Этот подход побуждает программистов создавать эффективные конвейеры разработки, поскольку интеграционные тесты на уровне кода не обязательно должны выполняться на первом этапе сборки — все зависит от времени, которое для них требуется. Так, в Maven модуль *Failsafe Plugin* запускает интеграционные тесты, используя команду `mvn failsafe:integration-test failsafe:verify`, после сборки проекта.

Разумеется, IDE поддерживает запуск и тестов `Test`, и тестов с другими соглашениями об именах.

В Gradle эта структура именования не учитывается. Для достижения той же цели в проектах Gradle используются несколько наборов тестовых источников, выполняемых по отдельности.

Интеграционные и системные тесты на уровне кода

Тесты на уровне кода — модульные, компонентные и интеграционные — обеспечивают быструю обратную связь в процессе разработки. Они позволяют программистам проверять, работает ли бизнес-логика так, как ожидалось, для изолированных компонентов и насколько разумно построена общая конфигурация.

Недостатки интеграционных тестов

Для того чтобы проверить, как ведет себя приложение в условиях эксплуатации, этих тестов недостаточно. Различия в технологии, конфигурации и времени выполнения в итоге приведут к пропускам в тестовых случаях. Это могут быть разные версии корпоративных контейнеров, несоответствия в конфигурации компонента после развертывания всего приложения, различные реализации баз данных или различия в сериализации JSON.

В конечном счете, приложение работает в среде эксплуатации. Поэтому имеет смысл проверить его поведение в средах, эквивалентных условиям эксплуатации.

Очевидно, стоит создать несколько тестовых областей для обоих тестов с изолированной областью и более быстрой обратной связью, а также интеграционных тестов. Недостатком интеграционных тестов на уровне кода является то, что часто они занимают много времени.

В прошлом в моих проектах интеграционные тесты, выполняемые в таких контейнерах, как Arquillian, обычно занимали большую часть времени сборки, и в результате она длилась десять минут и более. Это значительно замедляло конвейер непрерывной поставки кода, замедляло обратную связь и уменьшало количество сборок. Я пытался устранить этот недостаток, используя для тестов Arquillian удаленные и управляемые контейнеры. Их жизненный цикл дольше, чем тестовый прогон, что позволяет пренебречь временем запуска.

Интеграционные тесты на уровне кода — удобный способ быстро проверить конфигурацию приложения, которую нельзя протестировать с помощью модульных или компонентных тестов. Однако для тестирования бизнес-логики они неидеальны.

Интеграционные тесты, в ходе которых все приложение развертывается в имитируемых средах, таких как встроенные контейнеры, имеют определенное значение, но для проверки поведения в среде эксплуатации их недостаточно, поскольку они не эквивалентны условиям эксплуатации. Независимо от того, где протекает тестирование — на уровне кода или в имитируемых средах, — интеграционные тесты, как правило, замедляют выполнение всего конвейера.

Недостатки системных тестов

Наиболее полную проверку приложения обеспечивают системные тесты, при которых приложение полностью развертывается в условиях, подобных среде эксплуатации. Поскольку эти тесты выполняются на позднем этапе конвейера непрерывной поставки кода, обеспечиваемая ими обратная связь оказывается более медленной. Тестовые случаи, такие как проверка преобразований JSON в конечной точке HTTP, требуют больше времени на получение инженерами обратной связи.

Создание и обслуживание сложных тестовых сценариев занимает довольно много времени и требует значительных усилий. Корпоративные приложения нуждаются в определении и обслуживании тестовых данных и конфигурации, особенно когда задействовано много внешних систем. Например, в сквозном тесте, проверяющем создание автомобиля в приложении по производству автомобилей, нужно учесть такие внешние проблемы, как линия сборки, которая должна быть настроена так же, как и тестовые данные. Управление этими сценариями требует определенных усилий.

В сквозных тестах предпринимается попытка использовать внешние системы и базы данных аналогично тому, как это происходит в условиях эксплуатации. Это ставит задачу обслуживания недоступных сред или сред, содержащих ошибки. Недоступные внешние системы или базы данных вызывают сбой тестов, но приложение не отвечает за такие отказы. Этот сценарий нарушает требование предсказуемости, в соответствии с которым тесты не должны зависеть от внешних факторов, способных привести к ложным срабатываниям. Поэтому рекомендуется, чтобы системные тесты макетировали все внешние системы, которые не являются частью тестируемого приложения. Это позволяет строить предсказуемые сквозные тесты. Подробнее реализация этого подхода будет рассмотрена в подразделе «Системные тесты».

Выводы

Модульные и компонентные тесты на уровне кода проверяют изолированную конкретную бизнес-логику. Они обеспечивают немедленную обратную связь и позволяют быстро устранить случайные ошибки. В частности, компонентные тесты проверяют интеграцию программных модулей, связанных с бизнесом.

Ограничение компонентных тестов заключается в том, что они работают без имитационного контейнера, настраивая тестовые случаи программным способом. Интеграционные тесты основаны на инверсии управления, как в контейнерах приложений, где подключение компонентов требует от программистов меньших усилий. Однако разработка удобных для обслуживания тестовых случаев с использованием программного подхода по технологии модульных тестов в итоге обусловливает более эффективное тестирование. Далее в этой главе, в разделе «Обслуживание тестовых данных и сценариев», рассказывается, какие методы помогают создавать эффективные тестовые случаи.

Интеграционные тесты позволяют проверить техническую интеграцию, а также конфигурацию компонентов приложения. Их обратная связь, конечно, быстрее,

чем при развертывании приложения как части конвейера. Однако они не обеспечивают достаточной проверки по сравнению условиями эксплуатации.

Эти тесты — хороший первичный, базовый барьер, защищающий от распространенных или случайных ошибок. Поскольку запуск интеграционных тестов обычно занимает довольно много времени, имеет смысл ограничить их количество. В идеале тестовые фреймворки, такие как Arquillian, развертываются в управляемых или удаленных контейнерах, которые не останавливаются после завершения каждого тестового случая.

Системные тесты проверяют поведение приложения методами, более приближенными к условиям эксплуатации. Они обеспечивают возвращение максимально подробных сведений о том, работает ли все корпоративное приложение так, как ожидалось, включая бизнес-логику и технические компоненты. Для построения предсказуемых тестовых сценариев важно учитывать внешние обстоятельства, такие как базы данных и внешние системы.

Разработка тестовых случаев, а особенно сложных тестовых сценариев, требует много времени и значительных усилий. Вопрос в том, к чему стоит прилагать эти усилия. Чтобы проверить бизнес-логику и особенно сопряженные компоненты, рекомендуется задействовать компонентные тесты. Интеграционные тесты не обеспечивают исчерпывающей проверки, но все же требуют определенных времени и усилий. Имеет смысл использовать некоторые из них для быстрой обратной связи по поводу интеграции, но не для проверки бизнес-логики. Программисты могут также найти способы многократного применения созданных сценариев в нескольких тестовых областях, например в качестве интеграционных и системных тестов.

Общая цель должна заключаться в том, чтобы сократить время и усилия, затрачиваемые на разработку и обслуживание тестовых случаев, и таким образом свести к минимуму общее время выполнения конвейера и увеличить покрытие приложения тестами.

Системные тесты

Системные тесты реализуются для уже развернутого корпоративного приложения. Оно содержит тот же код, имеет ту же конфигурацию и то же время выполнения, что и в условиях реальной эксплуатации. Для инициирования тестовых случаев используются внешние способы связи, такие как HTTP. Цель тестов — проверить, что общий результат, такой как отклики по HTTP, состояние базы данных или связь с внешними системами, соответствует ожидаемым реакциям приложения.

Системные тесты отвечают на вопрос, что именно надо протестировать: приложение работает так же, как и в условиях эксплуатации, за исключением внешних систем, и доступ к нему можно получить с помощью обычных интерфейсов. Внешние системы имитируются, чтобы обеспечить предсказуемость тестов и возможность проверки коммуникации. В зависимости от сценария база данных может рассматриваться как часть приложения и использоваться аналогично или же заменяться макетом.

Управление тестовыми сценариями

Сценарии системных тестов часто становятся весьма сложными, затрагивая несколько задач и затрудняя понимание того, какой сценарий в действительности должен быть протестирован.

Чтобы упростить сценарии, имеет смысл сначала составить схему тестового случая без написания кода. Описание требуемых операций в комментариях или даже поначалу на бумаге позволяет составить ясное представление о том, что должен делать данный сценарий тестирования. Последующая фактическая реализация тестового сценария с учетом разумного уровня абстракции приведет к формированию более удобных тестовых случаев, которые можно использовать многократно. Мы рассмотрим этот подход в примере, который приводится далее.

Важно принимать во внимание тестовые данные. Чем бóльшая ответственность возлагается на сценарий, тем сложнее будет определять и обрабатывать тестовые данные. Имеет смысл приложить некоторые усилия, чтобы обеспечить функциональность тестовых данных, которые обычно используются в тестовых случаях. В зависимости от характера приложения и его предметной области иногда для этого даже выделяют специального инженера. Многоразовые и эффективные функции могут несколько упростить задачу, однако по-прежнему может потребоваться по крайней мере определить и описать в документации общие тестовые данные и сценарии.

В конечном счете не следует игнорировать сложность тестовых данных. Если предметная область приложения требует сложных сценариев, бессмысленно не принимать это во внимание, исключая определенные тестовые случаи или откладывая выполнение тестовых сценариев до тех пор, пока качество приложения не улучшится в процессе эксплуатации.

Для того чтобы создавать предсказуемые изолированные тестовые случаи, сценарий должен быть реализован по возможности автономно и без сохранения состояния. Тестовые случаи должны иметь начальную точку, приближенную к условиям эксплуатации, и не рассчитывать на определенное состояние системы. Следует учитывать, что параллельно с ними могут выполняться другие тесты и сценарии использования.

Например, при создании нового автомобиля нельзя делать предположения о количестве существующих автомобилей. В тестовом случае перед созданием нового автомобиля не следует проверять, пуст ли список автомобилей или то, что по окончании теста он содержит только один созданный автомобиль. Скорее нужно удостовериться, что этот автомобиль является частью общего списка.

По этой же причине стоит избегать влияния системных тестов на жизненный цикл среды выполнения. В ситуациях, связанных с внешними системами, необходимо управлять поведением макетов этих систем. Такие случаи должны быть, насколько это возможно, ограничены, чтобы одновременно могли выполняться и другие сценарии.

Моделирование внешних систем

Внешние системы используются в сценариях системных тестов так же, как и в среде эксплуатации. Однако, подобно макетированию объектов в модульных и компонентных тестах, системные тесты имитируют и макетируют внешние системы. Таким образом устраняются потенциальные проблемы, за которые приложение не несет ответственности. Системные тесты выполняются в специализированных средах, например предоставляемых фреймворками управления контейнерами. Тестируемый объект — обособленное приложение — развертывается, выполняется и конфигурируется так же, как и при эксплуатации.

Имитируемые внешние системы сконфигурированы так, чтобы обеспечить ожидаемое поведение при обращении к ним приложения. Подобно макетам объектов, они проверяют правильную связь в зависимости от сценария использования.

Для большинства сценариев применяемые базы данных не макетируются. При необходимости сценарий тестирования может управлять базой данных и записывать в нее данные как часть жизненного цикла теста. Если система управления базой данных выполняет множество задач, внешних по отношению к приложению, например содержит много кода базы данных или предоставляет поисковую систему, то, возможно, имеет смысл создать макет и имитировать это поведение.

Управление контейнером сильно упрощает эту задачу, позволяя абстрагировать системы как сервисы. Образы модулей можно заменить другими реализациями, не затрагивая тестируемое приложение. Макеты сервисов могут быть доступными и конфигурироваться из действующего системного теста, определяющего поведение и внешние тестовые данные (рис. 7.1).

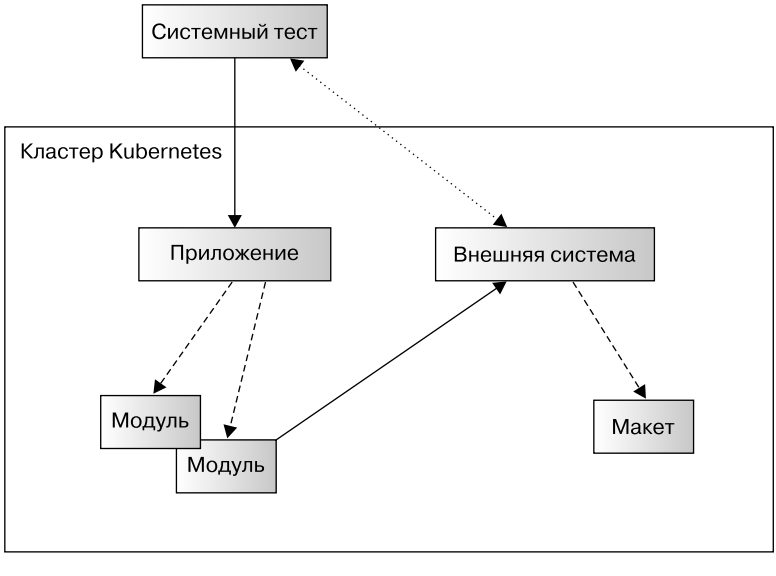

Рис. 7.1

Пунктирная линия на рис. 7.1 показывает направление управления и проверки макетируемой системы как части тестового сценария. Работающее приложение станет использовать внешнюю службу как обычно, с той разницей, что в действительности этот сервис заменен макетом.

Разработка системных тестов

Системные тесты выполняются как один из этапов в конвейере непрерывной поставки кода. Они подключаются к приложению, работающему в тестовой среде, вызывают бизнес-сценарии и проверяют результат.

Системные тестовые случаи обычно не влияют на жизненный цикл приложения. Приложение развертывается непосредственно в составе конвейера непрерывной поставки. При необходимости системные тесты контролируют состояние и поведение макетов внешних систем и содержимое баз данных.

Вообще говоря, имеет смысл разрабатывать системные тесты в виде отдельно собираемых проектов — так, чтобы код никак не зависел от основного проекта. Поскольку системные тесты получают доступ к приложению извне, они не должны влиять на то, как используется система. Системные тесты разрабатываются в соответствии с условиями конечных точек приложения. Также эти тесты не должны задействовать классы или функциональные возможности, которые являются частью приложения, например реализованные в приложении классы преобразования JSON. Определение технологии и доступа к системе извне, в виде отдельной сборки проекта, предотвращает нежелательные побочные эффекты, вызванные существующей функциональностью. Проект системного тестирования может находиться в том же репозитории, что и проект приложения.

В следующем примере будет построен системный тест по принципу нисходящего анализа, определяющий сценарии тестирования и соответствующие уровни абстракции.

Доступ к бизнес-сценариям приложения для производства автомобилей осуществляется через HTTP. Для работы приложения требуются обмен данными с внешними системами и доступ к базам данных. Для того чтобы протестировать процесс создания автомобиля, системный тест будет подключаться к работающему приложению так же, как это происходило бы в условиях реальной эксплуатации.

Для того чтобы управлять сценарием тестирования, тестовый случай создают с использованием логических операций, описанных в комментариях, а затем реализуют на нескольких уровнях абстракции:

```
public class CarCreationTest {

    @Test
    public void testCarCreation() {

        // убедиться, что машина 1234 не включена в список автомобилей

        // создать автомобиль
        //    с ID 1234
```

```
        //    и дизельным двигателем,
        //    красного цвета

        // убедиться, что у машины 1234
        //    дизельный двигатель
        //    и она красного цвета

        // убедиться, что машина 1234 есть в списке автомобилей

        // проверить инструкции конвейера сборки для машины 1234
    }
}
```

Комментарии описывают логические шаги, которые выполняются и проверяются при тестировании создания автомобиля. Они относятся к бизнес-области, а не к технической реализации.

Функции, соответствующие этим комментариям, реализуются в виде частных методов или, что еще лучше, делегатов. Последние инкапсулируют технические детали, а также потенциальное поведение жизненного цикла.

Создадим делегаты `CarManufacturer` и `AssemblyLine`, которые абстрагируют доступ и поведение приложений и делегатов. Они определяются как часть системного теста, не имеют отношения к одноименным управляемым компонентам и ничего не знают о них. Код проекта системного теста описан независимо от основного приложения. Он также может быть реализован с использованием другой технологии, сохраняя только совместимость с коммуникационным интерфейсом приложения.

В следующем фрагменте кода показана интеграция делегатов. Системный тест приложения для создания автомобиля содержит только бизнес-логику, связанную с реализацией, а действительные вызовы выполняются с помощью делегатов. Это позволяет создавать легко читаемые и обслуживаемые тестовые случаи. В таких системных тестах можно многократно использовать функциональность делегатов:

```
import javax.ws.rs.core.GenericType;

public class CarCreationTest {

    private CarManufacturer carManufacturer;
    private AssemblyLine assemblyLine;

    @Before
    public void setUp() {
        carManufacturer = new CarManufacturer();
        assemblyLine = new AssemblyLine();

        carManufacturer.verifyRunning();
        assemblyLine.initBehavior();
    }

    @Test
    public void testCarCreation() {
```

```
        String id = "X123A345";
        EngineType engine = EngineType.DIESEL;
        Color color = Color.RED;

        verifyCarNotExistent(id);

        String carId = carManufacturer.createCar(id, engine, color);
        assertThat(carId).isEqualTo(id);

        verifyCar(id, engine, color);

        verifyCarExistent(id);

        assemblyLine.verifyInstructions(id);
    }

    private void verifyCarExistent(String id) {
        List<Car> cars = carManufacturer.getCarList();
        if (cars.stream().noneMatch(c -> c.getId().equals(id)))
            fail("Автомобиль с id '" + id + "' не существует");
    }

    private void verifyCarNotExistent(String id) {
        List<Car> cars = carManufacturer.getCarList();
        if (cars.stream().anyMatch(c -> c.getId().equals(id)))
            fail(" Автомобиль с ID '" + id + "' уже существует");
    }

    private void verifyCar(String carId, EngineType engine, Color color) {
        Car car = carManufacturer.getCar(carId);
        assertThat(car.getEngine()).isEqualTo(engine);
        assertThat(car.getColor()).isEqualTo(color);
    }
}
```

Это простейший пример системного теста приложения. Делегаты, такие как `CarManufacturer`, выполняют низкоуровневый обмен данными и проверку:

```
public class CarManufacturer {

    private static final int STARTUP_TIMEOUT = 30;
    private static final String CARS_URI =
        "http://test.car-manufacture.example.com/" +
            "car-manufacture/resources/cars";

    private WebTarget carsTarget;
    private Client client;

    public CarManufacturer() {
        client = ClientBuilder.newClient();
        carsTarget = client.target(URI.create(CARS_URI));
    }

    public void verifyRunning() {
```

```
        long timeout = System.currentTimeMillis() + STARTUP_TIMEOUT * 1000;

        while (!isSuccessful(carsTarget.request().head())) {
            // ждать до истечения STARTUP_TIMEOUT,
            // затем сообщить о неудаче
            …
        }
    }

    private boolean isSuccessful(Response response) {
        return response.getStatusInfo().getFamily() ==
            Response.Status.Family.SUCCESSFUL;
    }

    public Car getCar(String carId) {
        Response response =
            carsTarget.path(carId).request(APPLICATION_JSON_TYPE).get();
        assertStatus(response, Response.Status.OK);
        return response.readEntity(Car.class);
    }

    public List<Car> getCarList() {
        Response response =
            carsTarget.request(APPLICATION_JSON_TYPE).get();
        assertStatus(response, Response.Status.OK);
        return response.readEntity(new GenericType<List<Car>>() {
        });
    }

    public String createCar(String id, EngineType engine, Color color) {
        JsonObject json = Json.createObjectBuilder()
                .add("identifier", id)
                .add("engine-type", engine.name())
                .add("color", color.name());

        Response response = carsTarget.request()
                .post(Entity.json(json));

        assertStatus(response, Response.Status.CREATED);

        return extractId(response.getLocation());
    }

    private void assertStatus(Response response, Response.Status
        expectedStatus) {
        assertThat(response.getStatus()).isEqualTo(expectedStatus.getStatusCode());
    }

    …
}
```

Делегат теста настроен для среды тестирования автомобилей. Эту конфигурацию можно сделать изменяемой, например, с помощью системного свойства

Java или переменной среды, чтобы обеспечить многократное использование теста в разных средах.

Если делегат должен включаться в жизненный цикл тестового случая, то его можно определить как правило JUnit 4 или модель расширения JUnit 5.

В данном примере тестовый случай подключается к работающему приложению для создания автомобилей через HTTP. Он может делать автомобили и читать список готовых автомобилей, преобразовывать и проверять ответы. Читатели могли обратить внимание на то, как делегат инкапсулирует внутренние коммуникации, такие как URL-адреса HTTP, коды состояния и JSON-преобразование. Его публичный интерфейс содержит только относящиеся к бизнес-области тестового сценария классы, такие как `Car` или `EngineType`. Типы сущностей предметной области, используемые в системных тестах, не должны соответствовать типам, определенным в приложении. По соображениям простоты в системных тестах могут применяться другие, более простые типы, которых достаточно для данного сценария.

Развертывание и управление внешними макетами

Мы только что узнали, как подключить системный тест к работающему корпоративному приложению. Но как контролировать и управлять внешней системой, которая используется в бизнес-сценарии приложения?

Внешние системы можно моделировать с помощью технологий макетного сервера, таких как *WireMock*. WireMock работает как автономный веб-сервер, настроенный так, чтобы определенным образом отвечать на конкретные запросы. Он действует как тестовый макет объекта на уровне кода, который имитирует и проверяет определенное поведение.

Преимущество применения фреймворков управления контейнерами для выполнения системных тестов заключается в том, что сервисы могут быть легко заменены макетными серверами. Инфраструктура внешней системы в виде конфигурации кода для среды системного тестирования может содержать образ Docker WireMock, который выполняется вместо реальной системы.

В следующем фрагменте кода показан пример конфигурации Kubernetes для системы конвейерной линии с использованием образа Docker WireMock в работающих модулях:

```
---
kind: Service
apiVersion: v1
metadata:
    name: assembly-line
    namespace: systemtest
spec:
    selector:
        app: assembly-line
    ports:
        - port: 8080
---
```

```
kind: Deployment
apiVersion: apps/v1beta1
metadata:
    name: assembly-line
    namespace: systemtest
spec:
    replicas: 1
    template:
        metadata:
            labels:
                app: assembly-line
        spec:
            containers:
            - name: assembly-line
                image: docker.example.com/wiremock:2.6
            restartPolicy: Always
---
```

Системный тест подключается к этому сервису, задействуя административный URL для настройки и изменения поведения макетного сервера.

В следующем фрагменте кода показана реализация тестового делегата `AssemblyLine` с использованием API WireMock для управления сервисом:

```
import static com.github.tomakehurst.wiremock.client.ResponseDefinitionBuilder.
okForJson;
import static com.github.tomakehurst.wiremock.client.WireMock.*;
import static java.util.Collections.singletonMap;

public class AssemblyLine {

    public void initBehavior() {
        configureFor("http://test.assembly.example.com", 80);

        resetAllRequests();

        stubFor(get(urlPathMatching("/assembly-line/processes/[0-9A-Z]+"))
                .willReturn(okForJson(singletonMap("status", "IN_PROGRESS"))));

        stubFor(post(urlPathMatching("/assembly-line/processes"))
                .willReturn(status(202)));
    }

    public void verifyInstructions(String id) {
        verify(postRequestedFor(urlEqualTo("/assembly-line/processes/" + id))
                .withRequestBody(carProcessBody()));
    }

    …
}
```

Начальное поведение таково, что экземпляр WireMock получает инструкцию, как правильно отвечать на HTTP-запросы. Поведение может быть изменено в процессе выполнения тестового случая, если необходимо представить более сложные процессы и варианты обмена данными.

Если более сложный тестовый сценарий подразумевает асинхронную связь, такую как длительные процессы, то для ожидания проверок в тестовых случаях может использоваться опрос.

Делегаты, созданные для изготовителя автомобилей и сборочной линии, могут многократно применяться в разных сценариях тестирования. В некоторых случаях может потребоваться выполнять системные тесты независимо друг от друга.

В разделе «Обслуживание тестовых данных и сценариев» мы увидим, какие еще методы и принципы помогают программистам разрабатывать удобные для обслуживания тестовые случаи.

Тесты производительности

Тесты производительности предназначены для проверки нефункционального требования — того, как система работает с точки зрения времени реакции. Они проверяют не бизнес-логику, а технологию, реализацию и конфигурацию приложения.

В условиях эксплуатации нагрузка на системы может сильно различаться. Это особенно актуально для общедоступных приложений.

Причины

Наряду с тестами, проверяющими бизнес-сценарии, бывает полезно проверить, обеспечивают ли приложение или его отдельные компоненты производительность, ожидаемую в среде эксплуатации. Причины этого — желание предотвратить падение производительности из-за ошибок, возникающих при повышенных нагрузках.

При построении сценариев тестирования производительности важно учитывать логику приложения. Одни вызовы запускают длительные процессы, другие — более короткие. Как правило, имеет смысл строить тесты производительности в соответствии с реалистичными сценариями эксплуатации с учетом частоты и характера запросов. Например, соотношение анонимных посетителей интернет-магазина и клиентов, фактически совершающих покупки, должно каким-то образом отражать реальное положение дел.

Имеет смысл также создавать тесты, в которых количество затратных по времени вызовов значительно превышает реальное, чтобы обнаружить потенциальные проблемы, которые могут возникнуть, если система попадет в условия экстремальных нагрузок.

В главе 9 мы увидим, почему тесты производительности в средах, отличных от реальной среды эксплуатации, плохо подходят для изучения ограничений и потенциальных узких мест приложения. Вместо того чтобы тратить большие усилия на разработку сложных сценариев тестирования производительности, имеет смысл вкладывать средства в получение технической информации о ре-

альных системах эксплуатации. Тем не менее мы рассмотрим несколько методов создания простых нагрузочных тестов, имитирующих для приложения условия высокой нагрузки, чтобы устранить наиболее явные проблемы.

Разумный подход состоит в том, чтобы имитировать обычную нагрузку, увеличив количество пользователей, обращающихся к приложению одновременно, и изучить, в какой момент приложение перестает отвечать на запросы. Если ответная реакция пропадает раньше, чем во время предыдущего тестового прогона, это может указывать на проблему.

Основные показатели производительности

Ключевые показатели производительности дают информацию о скорости реагирования приложения при нормальном поведении, а также при моделируемой рабочей нагрузке. Существует несколько регистрируемых показателей, непосредственно влияющих на пользователя, таких как время отклика или частота ошибок. Они описывают состояние системы и позволяют составить представление о ее поведении в ходе тестов производительности. Эти значения могут зависеть от количества пользователей, обращающихся к приложению одновременно, а также от тестового сценария.

Прежде всего надо понять, что время ответа приложения — это время, которое требуется для ответа на запрос клиента, включая все операции передачи данных. Оно напрямую влияет на качество предоставляемой услуги. Если время отклика превышает определенный порог, то могут возникнуть задержки, в результате которых запрос будет отменен или закончится неудачей. Второй показатель — это время ожидания, то время, которое проходит до получения сервером первого байта запроса. Оно зависит главным образом от параметров сети.

При тестировании производительности особенно интересно наблюдать, как изменяются время отклика и время ожидания по сравнению со средним значением. При увеличении нагрузки в какой-то момент приложение перестает отвечать на запросы. Такая ситуация может возникнуть по ряду причин. Например, могут быть заняты все доступные соединения или потоки, могут возникнуть задержки, может оказаться неудачной блокировка базы данных. Это описывает коэффициент ошибок запроса — процент неудавшихся запросов.

Количество параллельных пользователей или размер загрузки за определенный интервал времени влияют на показатели производительности приложения и должны учитываться в результатах теста. Чем больше пользователей, тем выше нагрузка на систему в зависимости от характера запроса. Это число связано с количеством параллельных транзакций, в том числе технических, и показывает, сколько одновременных транзакций может обрабатывать приложение.

Загрузка процессора и памяти дает представление о ресурсах приложения. Эти значения не всегда явно говорят о правильной работе приложения, однако показывают тенденцию потребления ресурсов при моделировании нагрузки.

Аналогично общая пропускная способность указывает на общий объем данных, которые сервер передает подключенным пользователям в каждый момент времени.

Ключевые показатели эффективности дают представление о скорости реагирования приложения. Они помогают собрать опытные данные и составить представление о тенденциях в процессе разработки. Этот опыт можно использовать для проверки будущих версий приложений. Тесты производительности могут указывать на потенциальное влияние изменений на производительность, особенно после внесения изменений в технологию, реализацию или конфигурацию.

Разработка тестов производительности

Создаваемые сценарии тестирования производительности должны быть близки к реальным условиям. Технология тестирования производительности должна поддерживать сценарии, которые не только увеличивают количество пользователей, но и имитируют их поведение. Типичным поведением может быть, например, посещение пользователем домашней страницы, вход в систему, переход по ссылке на статью, добавление товара в корзину и совершение покупки.

Существует несколько технологий тестирования производительности. На момент написания этой книги наиболее распространенными были *Gatling* и *Apache JMeter*.

Apache JMeter выполняет тестовые сценарии, которые ставят приложения в условия высокой нагрузки, и генерирует отчеты по результатам теста. Эта система использует конфигурацию на основе XML, поддерживает различные, в том числе нестандартные, протоколы связи и может применяться для воспроизведения записанных сценариев тестовой нагрузки. Apache JMeter определяет планы тестирования, которые содержат комбинации так называемых образцов и логических контроллеров. Они используются для определения тестовых сценариев, имитирующих поведение пользователя. Система JMeter распределяется посредством архитектуры «ведущий — ведомый» и может использоваться для генерации нагрузки из нескольких источников. Она имеет графический интерфейс для редактирования конфигурации плана тестирования. Инструменты командной строки позволяют проводить тестирование локально или на сервере непрерывной интеграции.

Система Gatling предлагает аналогичное решение для тестирования производительности, но в ней сценарии тестирования описываются программно, на Scala. Поэтому она обеспечивает большую гибкость при описании тестовых сценариев, поведения виртуальных пользователей и способов тестирования. Gatling также позволяет записывать варианты поведения пользователей и использовать их повторно. Поскольку тесты описываются программно, существует множество гибких решений, таких как динамическое получение тестовых случаев из внешних источников. Для проверки успешности выполнения тестовых запросов и всего тестового случая задействуются так называемые проверки и утверждения.

В отличие от JMeter Gatling работает на одном узле, а не в распределенной среде.

В следующем фрагменте кода показано описание простого моделирования для Gatling на Scala:

```
import io.gatling.core.Predef._
import io.gatling.core.structure.ScenarioBuilder
import io.gatling.http.Predef._
import io.gatling.http.protocol.HttpProtocolBuilder
import scala.concurrent.duration._

class CarCreationSimulation extends Simulation {

    val httpConf: HttpProtocolBuilder = http
    .baseURL("http://test.car-manufacture.example.com/car-manufacture/resources")
        .acceptHeader("*/*")

    val scn: ScenarioBuilder = scenario("create_car")
        .exec(http("request_1")
            .get("/cars"))
        .exec(http("request_1")
            .post("/cars")
            .body(StringBody("""{"id": "X123A234", "color": "RED",
                "engine": "DIESEL"}""")).asJSON
            .check(header("Location").saveAs("locationHeader")))
        .exec(http("request_1")
            .get("${locationHeader}"))

    pause(1 second)

    setUp(
        scn.inject(rampUsersPerSec(10).to(20).during(10 seconds))
            ).protocols(httpConf)
            .constantPauses

}
```

Сценарий `create_car` включает в себя три клиентских запроса, которые читают список всех автомобилей, создают автомобиль и подключаются к созданному ресурсу. Сценарии настраивают нескольких виртуальных пользователей. Количество пользователей начинается с десяти и увеличивается до 20 в течение десяти секунд.

Симуляция запускается через командную строку и выполняется в среде эксплуатации. Gatling представляет результаты тестов в файлах HTML. На рис. 7.2 показан результат тестирования Gatling в формате HTML для данного тестового примера.

Пример позволяет составить представление о возможностях тестов Gatling.

Поскольку тесты производительности должны отражать некие реалистичные пользовательские сценарии, имеет смысл повторно использовать для них уже существующие сценарии системного тестирования. Помимо программного описания поведения пользователей, предварительно записанные тестовые прогоны

могут применяться для передачи данных из внешних источников, таких как файлы журналов веб-сервера.

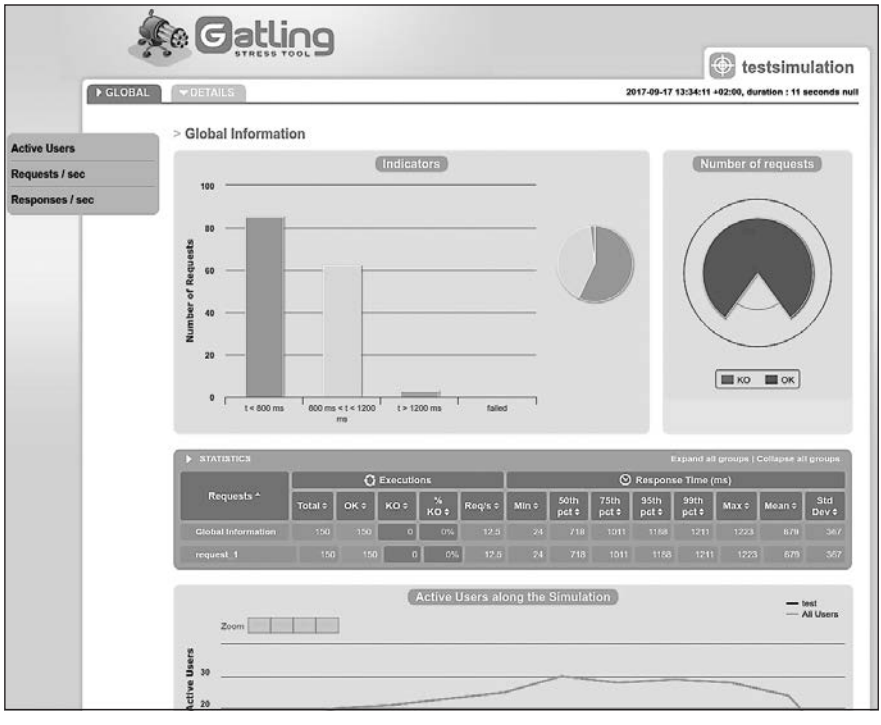

Рис. 7.2

Выводы

Цель тестирования производительности — не столько получение «зеленого» или «красного» результата, сколько идеи, собранные в ходе тестовых прогонов. В процессе прохождения тестов будут собраны отчеты о тестировании и поведении приложения. Эти наборы данных позволяют приобретать опыт и отслеживать тенденции в изменении производительности.

Тесты производительности могут проводиться автономно, однако они идеально работают непрерывно в рамках конвейера бесперебойной поставки кода. Иметь эти сведения без необходимости влиять на результат выполнения этапа конвейера полезно само по себе. После того как будут получены определенные результаты измерений, инженеры могут решить, следует ли считать прохождение теста неудачным, если измеренная производительность значительно снизилась по сравнению с обычными ожиданиями.

Это соответствует идее непрерывного улучшения или в данном случае отсутствия ухудшения времени отклика.

Локальное выполнение тестов

В главе 6 были рассмотрены процессы разработки и конвейеры непрерывной поставки кода. Для современных корпоративных приложений очень важно построить эффективный конвейер. Однако, несмотря на то что сервер непрерывной поставки выполняет все операции по сборке, тестированию и развертыванию, инженерам-программистам по-прежнему необходимо собирать и тестировать продукты в локальной среде.

Конвейеры непрерывной поставки кода в сочетании с надлежащими тестами довольно надежно гарантируют, что корпоративные приложения работают должным образом. Но если полагаться только на конвейер, то инженеры станут получать обратную связь позже и только после того, как изменения будут занесены в центральный репозиторий. Несмотря на то что именно эта идея лежит в основе непрерывной интеграции, программисты по-прежнему хотят быть уверенными в правильности изменений, прежде чем зафиксировать их в репозитории.

Фиксация в репозитории изменений, которые содержат случайные ошибки, мешает другим членам команды, так как понапрасну нарушает сборку. Ошибки, которые легко обнаружить, можно предотвратить, проверив приложение локально перед фиксацией изменений. Этого, очевидно, можно добиться с помощью тестов на уровне кода — модульных, компонентных и интеграционных, которые могут выполняться в локальных средах. Тестирование на уровне кода перед фиксацией изменений позволяет избежать большинства ошибок.

При разработке технических или сквозных задач, таких как перехватчики или преобразование JAX-RS JSON, инженеры также хотят получить обратную связь, прежде чем фиксировать изменения в конвейере. Как уже отмечалось, наиболее реалистичную проверку работающего приложения системные тесты обеспечивают.

Для того чтобы получить более быструю обратную связь, программисты могут писать сложные интеграционные тесты для локальных сред, работающие во встроенных контейнерах. Однако, как мы уже видели, для этого требуется довольно много времени и усилий и все равно не обеспечивает надежного тестового покрытия для всех случаев.

Использование контейнерных технологий позволяет инженерам запускать одни и те же программные образы в нескольких средах, в том числе локально. Существуют установки Docker для основных операционных систем. Контейнеры Docker можно запускать на локальных машинах так же, как и в среде эксплуатации, при необходимости создавая специфическую конфигурацию или свои сетевые соединения. Это позволяет реализовывать полноценные системные тесты в локальных средах. Несмотря на то что эта операция не обязательно должна выполняться в процессе разработки, она полезна для программистов, которые хотят проверить интеграционное поведение.

Программисты могут проводить операции сборки и тестирования локально так же, как в конвейере непрерывной поставки. Реализация этапов конвейера из командной строки значительно облегчает этот процесс. Команды `run` Docker позволяют динамически настраивать тома, сети и переменные среды на основе локального узла.

Для того чтобы автоматизировать этот процесс, отдельные команды сборки, развертывания и тестирования можно объединить в сценарии оболочки.

В следующем фрагменте кода показан пример такого сценария для Bash, выполняющего несколько операций. Bash-сценарии можно реализовывать и в Windows, используя эмуляторы Unix-консоли:

```
#!/bin/bash
set -e
cd hello-cloud/

# build
mvn package
docker build -t hello-cloud .

# deploy
docker run -d \
    --name hello-cloud-st \
    -p 8080:8080 \
    -v
$(pwd)/config/local/application.properties:/opt/config/application.properti
es \
    hello-cloud

# system tests
cd ../hello-cloud-st/
mvn test

# stopping environment
docker stop hello-cloud-st
```

Приложение hello-cloud находится в папке `hello-cloud/`, собрано с помощью Maven и Docker. Команда Docker `run` записывает конфигурацию в специальный файл свойств аналогично тому, как это было сделано в примере конфигурации фреймворка, показанном в главе 5.

В папке `hello-cloud-st/` содержатся системные тесты, которые выполняются для работающего приложения. Для того чтобы перенаправить системный тест в локальную среду, можно внести соответствующие изменения в конфигурацию hosts на локальном компьютере. Системные тесты проводятся посредством системы тестирования Maven.

Такой подход позволяет программистам проверить поведение приложения посредством полнофункциональных системных тестов, которые выполняются в конвейере непрерывной поставки, а при необходимости — локально.

Если для сценария системного тестирования требуется несколько внешних систем, то они также могут работать как контейнеры Docker аналогично тестовой

среде. Приложения, работающие в среде управления контейнерами, используют логические имена сервисов для распознавания внешних систем. Это можно сделать и для загруженных контейнеров Docker, которые являются частью специализированных сетей Docker. Docker распознает имена контейнеров внутри контейнеров, работающих в одной сети.

Этот подход используется для локального запуска всех видов сервисов и особенно полезен для запуска макетных серверов.

В следующем фрагменте показан пример создания локальной тестовой среды:

```
#!/bin/bash
# previous steps omitted

docker run -d \
    --name assembly-line \
    -p 8181:8080 \
    docker.example.com/wiremock:2.6

docker run -d \
    --name car-manufacture-st \
    -p 8080:8080 \
    car-manufacture

# …
```

Как и в случае с примером системного теста, сервер WireMock конфигурируется как часть тестового случая. Локальная среда должна гарантировать, что имена узлов указывают на соответствующие локальные контейнеры.

Для более сложных конфигураций имеет смысл запускать сервисы в кластере управления контейнерами. Существуют локальные варианты установки Kubernetes и OpenShift. Управление контейнерами абстрагирует узлы кластера. Поэтому для определения инфраструктуры как кода не имеет значения, работает кластер локально, как отдельный узел, в среде сервера локально или в облаке. Это позволяет инженерам применять те же самые определения, что и в тестовых средах. Запуск локальной установки Kubernetes упрощает сценарии оболочки до нескольких команд `kubectl`.

Если локально установленные Kubernetes или OpenShift занимают слишком много места, можно использовать такие упрощенные альтернативы для управления кластерами, как Docker Compose. Эта система также определяет многоконтейнерные среды и их конфигурацию в виде файлов с описанием инфраструктуры как кода, исполняемых одной командой. Она обеспечивает примерно те же возможности, что и Kubernetes. Более сложный способ организовать и запустить контейнеры Docker — Arquillian Cube.

Локальная автоматизация этапов контейнера с помощью сценариев повышает продуктивность работы программиста. Выполнение системных тестов на локальных машинах помогает инженерам получить более быструю обратную связь с меньшими потерями.

Обслуживание тестовых данных и сценариев

Тестовые случаи позволяют убедиться, что при развертывании в условиях эксплуатации приложение будет вести себя так, как ожидалось. Тесты также гарантируют, что эти ожидания сохранятся по мере разработки новых функций.

Однако недостаточно один раз описать тестовые сценарии и тестовые данные. Со временем бизнес-логика будет развиваться и меняться, и тестовые случаи должны адаптироваться к этим изменениям.

Насколько важны обслуживаемые тесты

И для написания тестовых случаев, и для управления ими крайне важно создать обслуживаемый тестовый код. Со временем количество тестовых случаев будет увеличиваться. Для того чтобы сохранять продуктивность разработки, необходимо потратить некоторое время и приложить определенные усилия для получения качественного тестового кода.

В отношении кода самого приложения любой инженер согласится с тем, что качество кода является важным требованием. Однако, поскольку тесты не являются частью приложения, которая работала бы в условиях эксплуатации, к ним часто относятся иначе. Опыт показывает, что программисты редко тратят время и прилагают усилия, чтобы улучшить качество тестового кода. Однако качество тестовых случаев имеет огромное влияние на продуктивность работы программиста.

О том, что тесты написаны плохо, говорит ряд признаков.

Признаки недостаточного качества тестов

Вообще говоря, если в процессе разработки на тестовый код затрачивают больше времени, чем на код собственно приложения, это может быть признаком плохо построенных или разработанных тестов. Каждая новая или измененная функция может вызывать сбой некоторых тестов. Насколько быстро тестовый код адаптируется к изменениям? Какое количество тестовых данных и функций необходимо изменить? Насколько легко добавлять тестовые случаи в существующую кодовую базу?

Если неудачные тесты снабжаются аннотацией `@Ignored` на сколько-нибудь продолжительное время, это тоже признак потенциально недостаточного качества тестов. Если тестовый случай логически все еще нужен, его необходимо стабилизировать и исправить, если же устарел — удалить. Однако никогда не следует удалять тесты, чтобы сэкономить время и усилия, необходимые для их исправления, если тестовые сценарии все еще логически важны.

Копирование и вставка тестового кода также являются тревожным сигналом. К сожалению, эта практика довольно широко распространена в корпоративных

проектах, особенно когда поведение тестовых сценариев различается незначительно. Копирование и вставка нарушают принцип *DRY* (don't repeat yourself — «не повторяй сам себя») и приводит к значительному дублированию кода, что усложняет последующие изменения.

Качество тестового кода

Для обеспечения постоянной скорости разработки важно не только качество кода самого приложения, но и качество тестового кода. Однако к тестам, как правило, относятся не так, как к основному коду. Опыт показывает, что при создании корпоративных проектов редко инвестируют время и усилия в реструктуризацию.

В общем случае к обеспечению высокого качества тестового кода применимы те же методы, которые действуют и для основного кода приложения. Однако некоторые принципы для тестов особенно значимы.

Прежде всего, безусловно, важен принцип DRY. На уровне кода это означает: избегать повторения определений, тестовых процедур и дублирования кода, имеющего незначительные различия.

Этот принцип применим и для тестовых данных. Как показывает опыт, существование нескольких сценариев тестовых случаев, использующих аналогичные тестовые данные, побуждает разработчиков задействовать метод копирования и вставки. Однако после внесения изменений в тестовые данные это приводит к созданию неудобной в обслуживании кодовой базы.

Это касается также проверки утверждений и макетов. Операторы контроля и проверки, которые задействуются один за другим в методе тестирования, аналогичным образом приводят к дублированию и сложностям в обслуживании.

Как правило, самой большой проблемой качества тестового кода является отсутствие уровней абстракции. Тестовые случаи слишком часто содержат разные показатели и задачи. В них смешиваются бизнес-сценарии и технические задачи.

Позвольте привести пример плохо написанного системного теста в виде псевдокода:

```
@Test
public void testCarCreation() {
    id = "X123A345"
    engine = EngineType.DIESEL
    color = Color.RED

    // убедиться, что машины X123A345 не существует
    response = carsTarget.request().get()
    assertThat(response.status).is(OK)
    cars = response.readEntity(List<Car>)
    if (cars.stream().anyMatch(c -> c.getId().equals(id)))
        fail("Машина с ID '" + id + "' уже существует")

    // создать машину X123A345
    JsonObject json = Json.createObjectBuilder()
```

```
            .add("identifier", id)
            .add("engine-type", engine.name())
            .add("color", color.name())

    response = carsTarget.request().post(Entity.json(json))
    assertThat(response.status).is(CREATED)
    assertThat(response.header(LOCATION)).contains(id)

    // проверить машину X123A345
    response = carsTarget.path(id).request().get()
    assertThat(response.status).is(OK)
    car = response.readEntity(Car)
    assertThat(car.engine).is(engine)
    assertThat(car.color).is(color)

    // убедиться, что машина X123A345 существует

    // ... аналогичные вызовы

    if (cars.stream().noneMatch(c -> c.getId().equals(id)))
        fail("Машины с ID '" + id + "' не существует");
}
```

Читатели, возможно, заметили: чтобы разобраться в этом тестовом случае, требуется довольно много усилий. Комментарии немного помогают, но они скорее являются признаком плохо построенного кода.

Однако этот пример похож на созданный ранее пример системного теста.

Проблема таких тестовых случаев не только в том, что их сложнее понять. Смешение в одном тесте нескольких технических задач и бизнес-сценариев, объединенных в одном классе или даже в одном методе, приводит к дублированию и исключает то, что их будет удобно обслуживать. Что произойдет, если изменится полезная нагрузка сервиса по производству автомобилей? А если изменится логическая последовательность операций в тестовом случае? Что делать, если понадобится написать новые тестовые случаи с похожей логической последовательностью, но разными данными? Придется ли программистам копировать и вставлять весь код, изменяя в нем лишь несколько деталей? А если изменится способ обмена данными и это будет уже не HTTP, а другой протокол?

Для тестовых случаев самым важным критерием качества кода является применение правильных уровней абстракции в сочетании с делегированием.

Программистам следует спросить себя о целях этого тестового сценария. В нем существует последовательность логических операций, поэтапно проверяющая, как создается автомобиль. Есть коммуникационная часть, включающая в себя вызовы через HTTP и преобразования JSON. Также может быть задействована внешняя система, представленная, например, как макетный сервер, которым необходимо управлять. И есть утверждения и проверки, которые нужно проделать с учетом всех этих аспектов.

Именно по этой причине предыдущий пример системного теста состоит из нескольких компонентов, каждый из которых выполняет свои обязанности. Должен быть один компонент для доступа к тестируемому приложению, включая

все необходимые детали для реализации связи. В примере за это отвечал делегат производства автомобилей.

Подобно делегату конвейерной линии, имеет смысл создать по компоненту для каждой макетной системы. Эти компоненты инкапсулируют конфигурацию, управление и проверку макетных серверов.

Проверки, выполняемые на уровне тестирования бизнес-сценариев, также целесообразно передать либо в частные методы, либо делегатам, в зависимости от ситуации. Затем делегаты теста могут инкапсулировать логику, создав дополнительные уровни абстракции, если этого требует технология или тестовый случай.

Все эти классы и методы делегатов становятся едиными точками ответственности. Их можно многократно использовать во всех аналогичных тестовых случаях. Потенциальные изменения влияют только на точки ответственности, не затрагивая другие части тестовых случаев. Для этого требуется создать ясные интерфейсы между компонентами, которые скрывали бы детали реализации. По этой причине имеет смысл, особенно для системных тестов, иметь выделенное простое представление модели. Эта модель может быть реализована просто и понятно, возможно, с меньшей безопасностью типов, чем в основном коде приложения.

Разумный подход разработки с нуля, аналогичный примененному в предыдущем примере системного теста, заключается в том, чтобы сначала написать комментарии, а затем по мере перехода на более глубокие уровни абстракции заменять их делегатами. Сначала создается логическая структура теста, а затем добавляются детали реализации. Придерживаясь этого подхода, мы естественным образом избегаем смешивания бизнес-сценариев и технических задач при тестировании. Он также позволяет упростить интеграцию технологии тестирования, поддерживающей эффективное написание тестов, такой как *Cucumber-JVM* или *FitNesse*.

Поддержка технологий тестирования

Некоторые технологии тестирования также поддерживают создание тестов, удобных для обслуживания. Например, AssertJ позволяет создавать нестандартные утверждения. В нашем тестовом случае нужно проверить правильность установленного двигателя и цвета автомобиля — параметров, инкапсулированных в его технические характеристики. Пользовательские утверждения позволяют уменьшить дублирование в области тестирования.

Далее показано специальное утверждение `AssertJ` для проверки автомобиля:

```
import org.assertj.core.api.AbstractAssert;

public class CarAssert extends AbstractAssert<CarAssert, Car> {

    public CarAssert(Car actual) {
        super(actual, CarAssert.class);
    }

    public static CarAssert assertThat(Car actual) {
```

```
        return new CarAssert(actual);
    }

    public CarAssert isEnvironmentalFriendly() {
        isNotNull();

        if (actual.getSpecification().getEngine() != EngineType.ELECTRIC) {
            failWithMessage("Expected car with environmental friendly
                engine but was <%s>",
                    actual.getEngine());
        }

        return this;
    }

    public CarAssert satisfies(Specification spec) {
        …
    }

    public CarAssert hasColor(Color color) {
        isNotNull();

        if (!Objects.equals(actual.getColor(), color)) {
            failWithMessage("Expected car's color to be <%s> but was <%s>",
                    color, actual.getColor());
        }

        return this;
    }

    public CarAssert hasEngine(EngineType type) {
        …
    }
}
```

Это утверждение можно использовать в пределах области тестирования. Для метода `assertThat()` должен быть выбран правильный статический импорт класса `CarAssert`:

```
assertThat(car)
        .hasColor(Color.BLACK)
        .isEnvironmentalFriendly();
```

Приведенные в этой главе примеры тестов, за исключением встроенных контейнеров приложений и Gatling, написаны в основном на Java, JUnit и Mockito. Существуют десятки других тестовых технологий, в которых используются различные фреймворки, а также динамические JVM-языки.

Один из известных примеров такого фреймворка — *Spock Testing Framework*, в котором используется Groovy. Причины применения этой технологии — возможность писать более компактные и удобные для обслуживания тесты. Поскольку динамические JVM-языки, такие как Groovy и Scala, лаконичнее, чем простой Java, эта идея представляется разумной.

Тестовые фреймворки, такие как Spock, действительно позволяют создавать тестовые случаи, требующие минимального кода. В них применяются возможности динамического JVM-языка, такие как меньшие ограничения на имена методов, например `def "car X123A234 should be created"()`. При тестировании в Spock обеспечиваются также легко читаемые тесты.

Однако читаемые тесты можно получить при использовании любой технологии тестирования, если уделять внимание качеству тестового кода. В частности, удобство обслуживания больше зависит от хорошо продуманных тестовых случаев и правильно выбранных уровней абстракции, чем от задействованной технологии. Когда тестовые примеры становятся слишком сложными, влияние технологии на удобство обслуживания становится менее заметным.

При выборе технологии тестирования следует также учитывать опыт работы с ней команды. На момент написания этой книги разработчики корпоративных Java-приложений, как правило, не были знакомы с динамическими JVM-языками.

Однако качество тестового кода важнее, чем используемая технология. Применять рекомендованные стандарты разработки программного обеспечения к написанию тестов следует обязательно, а задействовать другие тестовые фреймворки — нет. Реструктуризация тестовых случаев часто позволяет сделать тесты более удобными в обслуживании, позволяет многократно использовать компоненты теста и в конечном счете повышает качество программного продукта.

Резюме

Тесты необходимы для проверки функциональности программного обеспечения в имитируемых средах. Тестирование программного обеспечения должно быть предсказуемым, изолированным, надежным, быстрым и автоматизированным. Для того чтобы обеспечить эффективный жизненный цикл проекта, важно, чтобы тесты были удобными в обслуживании.

Модульные тесты проверяют поведение отдельных модулей приложения, в основном отдельных сущностей, контуров и классов управления. Компонентные тесты проверяют, как ведут себя сопряженные компоненты. Интеграционные тесты удовлетворяют потребность в проверке взаимодействия компонентов Java EE. Интеграционные тесты базы данных используют встроенные базы данных и автономные JPA для проверки преобразования данных в процессе их сохранения. Системные тесты проверяют развернутые приложения, работающие в реальных средах. Управление контейнерами в значительной степени поддерживает работающие среды системного тестирования благодаря применению макетных приложений.

Для того чтобы проверить функциональность до того, как она будет перенесена в центральный репозиторий, инженеры должны иметь возможность выполнять тесты в локальной среде. Изменения, которые содержат случайные ошибки, мешают их товарищам по команде, нарушая сборку. Docker, Docker Compose

и Kubernetes могут работать в локальных средах, что позволяет программистам заблаговременно проверять поведение приложения. Целесообразно создавать простые сценарии автоматизации, включающие в себя соответствующие операции.

Для того чтобы обеспечить постоянную скорость разработки, необходимо создавать тестовые случаи, удобные для обслуживания. В общем случае тестовый код должен быть таким же качественным, как и код основного приложения. Это требует реструктуризации, правильного выбора уровней абстракции и высокого качества программного обеспечения в целом.

Эти принципы, в сущности, более полезны, чем внедрение сложных тестовых фреймворков с использованием динамических JVM-языков. Такие фреймворки, как Spock, позволяют создавать легко читаемые лаконичные тестовые случаи. Но применение рекомендованных стандартов разработки программного обеспечения более положительно влияет на общее качество тестового кода, особенно по мере усложнения тестовых сценариев. Независимо от используемой технологии тестирования инженерам-программистам рекомендуется уделять внимание качеству тестового кода, чтобы поддерживать работоспособность тестовых случаев.

Следующая глава посвящена распределенным системам и архитектурам микросервисов.

8 Микросервисы и системная архитектура

В предыдущих главах описывалась разработка отдельных корпоративных приложений на Java EE. Современные приложения содержат определения инфраструктуры и конфигурации как кода, что позволяет создавать среды в автоматическом режиме как на локальных, так и на облачных платформах. Конвейеры непрерывной поставки кода в сочетании с достаточными автоматизированными тестовыми случаями позволяют поставлять корпоративные приложения с высокими качеством и производительностью. Этому способствует принятый в Java EE современный принцип нулевой зависимости.

Корпоративные системы редко поставляются с единственным набором задач, который может быть разумно реализован в виде одного корпоративного приложения. Традиционно корпоративные приложения объединяют в себе несколько аспектов бизнеса в виде монолитного продукта. Но насколько целесообразно использовать этот подход при создании распределенных систем?

В этой главе будут рассмотрены следующие темы.

- Причины создания распределенных систем.
- Возможности и проблемы распределенных систем.
- Как создавать взаимозависимые приложения.
- Контуры приложений, API и документация.
- Согласованность и масштабируемость: проблемы и решения.
- Регистрация событий, событийно-ориентированные архитектуры и CQRS.
- Архитектуры микросервисов.
- Java EE в мире микросервисов.
- Реализация эластичной коммуникации.

Причины создания распределенных систем

Один из первых вопросов, которые следует задать себе перед проектированием распределенной системы: зачем она нужна? Есть несколько технических причин создания распределенных систем.

Типичные корпоративные сценарии, по существу, являются распределенными. Пользователи или другие системы, находящиеся в разных местах, должны взаимодействовать с сервисом. Взаимодействие осуществляется по сети.

Вторая причина — масштабируемость. Если одиночное приложение достигает предела, из-за чего не может дальше надежно обслуживать общую нагрузку клиентов, необходимо распределить бизнес-логику между несколькими узлами.

Аналогичные рассуждения можно привести и относительно отказоустойчивости системы. Одиночное приложение представляет собой единую точку отказа, если оно недоступно, то клиенты не смогут пользоваться сервисом. Распределение сервисов между несколькими точками увеличивает доступность и устойчивость.

Есть и другие, не технологические причины. Приложение выполняет определенные бизнес-задачи. На языке проблемно-ориентированного проектирования они определяются *ограниченным контекстом* приложения. Ограниченные контексты включают в себя бизнес-задачи, логику и данные приложения и отделяют их от внешних задач.

Подобно тому как инженеры оформляют код кластерных задач в виде пакетов и модулей, имеет смысл создавать аналогичные контексты на уровне систем. Сопряженная бизнес-логика и функциональность группируются в виде отдельных сервисов как часть отдельных приложений. Данные и схема также являются частью ограниченного контекста. Поэтому он может быть инкапсулирован в нескольких сущностях базы данных, принадлежащих соответствующим распределенным приложениям.

Проблемы распределенных систем

Учитывая все преимущества распределенных систем, особенно технологические, такие как масштабируемость, можно задать вопрос: почему инженеры не делают все приложения распределенными? Дело в том, что за выгоды, которые несут распределенные системы, приходится платить определенными накладными расходами.

Как правило, суммарные издержки, превышающие собственно бизнес-логику системы, умножаются на количество задействованных приложений. Например, если для одиночного монолитного приложения требуется мониторинг, то при его распределении все приложения, полученные в результате распределения, будут также нуждаться в мониторинге.

Потери за счет пропускной способности

Главные накладные расходы в распределенных средах приходятся на обмен данными между системами.

Обмен данными в рамках одного процесса очень эффективен. При вызове функций, которые являются частью приложения, накладных расходов практически нет. Но как только требуется коммуникация между процессами или удаленная коммуникация, инженеры должны определить абстракции интерфейса. Необходимо определить протоколы связи, такие как HTTP, и использовать их для обмена информацией.

Это требует определенных времени и усилий. Коммуникация между приложениями должна определяться, внедряться и поддерживаться. В рамках одного приложения она сводится к вызовам методов.

Коммуникацию приходится учитывать также при разработке бизнес-сценариев. Уже нельзя предполагать, что функции или данные могут быть применены без каких-либо накладных расходов. Коммуникация в распределенной системе входит в круг задач приложения.

Потери производительности

При переходе к распределенным приложениям падает производительность всех систем. Обмен данными в компьютерной сети происходит медленнее, чем внутри одного узла. Поэтому в сети всегда будут возникать определенные накладные расходы.

Потери производительности обусловлены не только самой коммуникацией, но и необходимостью синхронизации. Даже в рамках одного процесса синхронизация требует определенного времени процессора, в распределенных средах это влияние намного больше.

Однако, несмотря на эти потери, распределенные среды в итоге увеличивают общую производительность системы по мере масштабирования приложения. Горизонтальное масштабирование всегда сопровождается определенными потерями производительности по сравнению с одним экземпляром приложения.

Организационные расходы

Распределенные системы, содержащие несколько приложений, безусловно, требуют бóльших организационных усилий, чем одно приложение.

Несколько приложений подразумевают несколько развертываний, которыми необходимо управлять. Развертывание новых версий может повлиять на зависимые приложения. Команда разработчиков должна гарантировать, что версии развертываемых приложений хорошо совместимы. У одиночных монолитных приложений нет этой проблемы, поскольку их согласованность — внутренняя.

Кроме того, каждое приложение разрабатывается в рамках отдельного проекта, хранится в своем репозитории, и, как правило, за него отвечает определенная команда разработчиков. Наличие нескольких команд требует обмена информацией, не столько технического, сколько чисто человеческого. Точно так же, как при развертывании приложений, должны быть согласованы обязанности, границы системы и зависимости.

Как разрабатывать системные среды

Несмотря на все эти проблемы и издержки, многие сценарии по-прежнему требуют распределенной среды. Важно отметить, что для построения распределенных систем должны быть веские причины, так как это связано с затратами и рисками. Если в таких системах нет необходимости, следует отдавать предпочтение разработке монолитных приложений.

Теперь рассмотрим, как строить разумные системные среды, адаптированные к требованиям бизнеса.

Карты контекстов и ограниченные контексты

Ограниченные контексты определяют круг задач приложения в бизнес-логике, поведении и владении данными. Так называемые *карты контекстов*, описанные в концепциях проблемно-ориентированного программирования, представляют собой всю системную среду. В ней собраны индивидуальные задачи, контексты и активы всех приложений. Таким образом, карта контекстов показывает, как ограниченные контексты обмениваются информацией между собой.

Далее представлена карта контекстов для предметной области «автомобили», включающая в себя два ограниченных контекста (рис. 8.1).

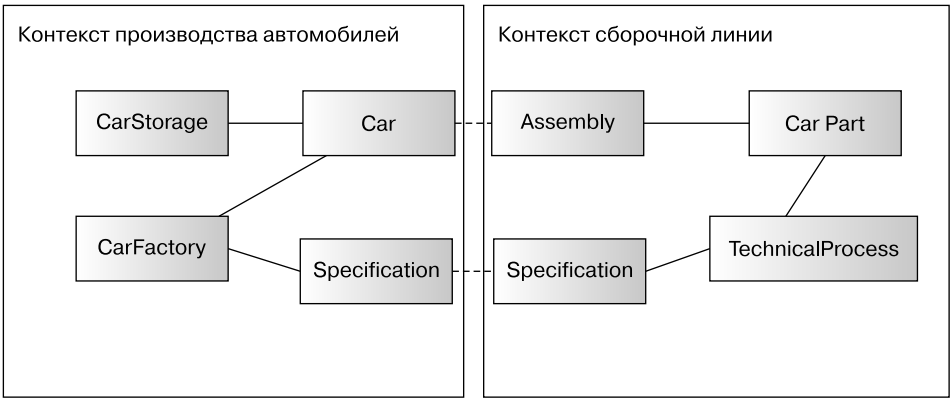

Рис. 8.1

Рекомендуется рассмотреть различные задачи системы и только потом проектировать и делить их на приложения. Отсутствие ясности в отношении круга задач, решаемых приложением, обычно проявляется сразу же, как только будет записана контекстная карта системы.

Контекстные карты полезны не только при первоначальном описании проекта, но и при пересмотре и уточнении его задач после изменения бизнес-функций. Для того чтобы границы и круг обязанностей распределенных приложений не размывались, рекомендуется время от времени их пересматривать.

Разделение задач

Задачи приложения, а также их отличия от других приложений должны быть четко описаны.

Точно так же, как и на уровне кода, должны быть разделены задачи нескольких приложений. Здесь применяется принцип единой ответственности.

В число задач приложения входят все коммерческие задачи, контуры приложений и данные, принадлежащие приложению. По мере развития и изменения бизнес-логики эти задачи следует периодически пересматривать. Это может привести к разделению приложения или, наоборот, объединению нескольких приложений в одно. Обязанности и задачи, вытекающие из контекстной карты, должны отражаться в системных приложениях.

Важными аспектами распределенных приложений являются данные и принадлежность данных. Бизнес-процессы, будучи частью ограниченного контекста, определяют данные, используемые в бизнес-сценариях. Те из них, что принадлежат определенному приложению, входят в круг его задач и доступны только через определенные контуры. Бизнес-сценарии, требующие доступа к данным, которые принадлежат другому, удаленному приложению, должны получать информацию удаленно, вызывая соответствующие бизнес-сценарии.

Рабочие команды

Распределение по командам и организационная структура — важные факторы, которые следует учитывать при проектировании распределенных систем, поскольку, по крайней мере на момент написания книги программное обеспечение создавалось людьми. Учитывая закон Конвея, согласно которому разработанная система в конечном счете повторяет структуру связей внутри организации, все рабочие команды должны быть соответствующим образом описаны в системе в виде приложений.

Другими словами, имеет смысл сделать так, чтобы каждое приложение разрабатывалось отдельной командой. Иногда в зависимости от задач и размеров приложений одна команда может создавать их несколько.

Здесь, как и при разработке структуры кода проекта, применяется подход, подобный использованному при создании горизонтальной и вертикальной

структуры модулей. Как структуры модулей строятся в соответствии с требованиями бизнеса, так и команды разработчиков организованы вертикально, отражая структуру карты контекстов. Например, вместо того, чтобы создавать несколько групп экспертов по архитектуре, разработке или операциям программного обеспечения, можно создать команды по *производству автомобилей*, *сборочной линии* и *управлению заказами*.

Жизненные циклы проектов

Поскольку в распределенной системе каждая команды разрабатывает свое приложение, у каждого из этих приложений будет собственный жизненный цикл проекта. Он включает в себя, в частности, то, как работают команды — например, как они организуют свои sprint-циклы.

Циклы и схемы развертывания также определяются жизненным циклом проекта. Для того чтобы система в целом оставалась последовательной и функциональной, необходимо определить потенциальные зависимости от развертывания других приложений. От этого зависит не только доступность приложения.

Развернутые версии приложений должны быть совместимыми. Для того чтобы это обеспечить, следует четко описать взаимозависимые приложения в карте контекстов. Команды разработчиков должны тесно общаться при внесении изменений в зависимые сервисы.

И здесь снова приходит на помощь карта контекстов: составление четкой карты контекстов, содержащей ограниченные контексты, помогает определить взаимозависимые приложения и их обязанности.

Как разрабатывать системные интерфейсы

После того как определены задачи системной среды, необходимо указать границы зависимых систем.

В предыдущих главах мы рассмотрели различные протоколы связи и способы их реализации. Возникает вопрос: как разрабатывать интерфейсы приложений? Какие факторы необходимо учитывать, особенно в распределенных системах?

Что надо учитывать при разработке API

Место каждого приложения в системе определяется его бизнес-логикой.

Соответственно, API приложения должен представлять эту бизнес-логику. Открытый API представляет бизнес-сценарии, выполняемые определенным приложением. Это означает, что эксперт в данной области бизнеса может определить бизнес-сценарии по API, без каких-либо дополнительных технических знаний.

В идеале бизнес-сценарии должны быть представлены в виде простых и понятных интерфейсов. Вызов бизнес-сценария не должен требовать дополнительных

технических операций или передачи технических деталей, не являющихся частью бизнес-логики. Например, если бизнес-сценарий «создать автомобиль» может быть вызван в виде одной операции, то API приложения по производству автомобилей не должен требовать нескольких вызовов, содержащих технические детали.

API должен абстрагировать бизнес-логику в виде простого и ясного интерфейса. Поэтому он не должен быть связан с реализацией приложения. Реализация интерфейса должна быть независимой от выбранной технологии. Это также означает, что выбранный формат связи не должен накладывать слишком много ограничений на используемую технологию.

Исходя из сказанного, имеет смысл выбрать такую технологию, которая опиралась бы на стандартные протоколы, такие как HTTP. Инженер, разрабатывающий приложение, скорее всего, знает все основные протоколы, поскольку они поддерживаются различными технологиями и инструментами. Создание интерфейсов приложений в виде веб-сервисов HTTP позволяет разрабатывать клиентскую часть на базе любой технологии, поддерживающей HTTP.

Абстрагирование бизнес-логики в виде ясных и простых интерфейсов, основанных на стандартных протоколах, также позволяет изменять используемые в приложении реализации, технологии и платформы. Если приложение Java Enterprise реализовано только в виде HTTP-сервиса, то его технологию можно заменить другой реализацией и это не потребует изменения зависимых от него клиентов.

Управление интерфейсами

Интерфейсы приложений часто меняются в процессе разработки. Появляются новые бизнес-сценарии и изменяются уже существующие. Как эти изменения отражаются в API?

Это зависит от природы и среды корпоративного приложения, от того, насколько стабильным должен быть API. Если команда разработки проекта отвечает также за его обслуживание, всех клиентов и их жизненные циклы, то в API можно вносить произвольные изменения, одновременно перенося их в клиентскую часть. То же самое касается ситуации, когда по какой-то причине жизненные циклы всех задействованных приложений идентичны.

Однако обычно жизненные циклы приложений в распределенных системах не так тесно связаны. Для любой другой модели «клиент/сервер» или для приложений, имеющих разные жизненные циклы, API не должны нарушать работу существующих клиентов. Это означает, что API должны обеспечивать полную обратную совместимость, не внося критических изменений.

Устойчивые к изменениям API

Существуют определенные принципы разработки интерфейсов, которые предотвращают лишние сбои. Например, введение новых, необязательных исходных данных не должно мешать выполнению команды. Технология должна быть устойчивой, и работа должна продолжаться, если предоставлены все необходи-

мые данные. Это соответствует принципу: *быть консервативным в том, что вы делаете, и либеральным в том, что принимаете.*

Поэтому добавлять новые, необязательные функции и данные должно быть возможно без нарушения работы клиентов. Но как быть, если меняется сама логика?

Изменение бизнес-логики

Вопрос, который необходимо задать себе в этот момент: что означает изменение API с точки зрения бизнес-сценария? Поведение приложения больше не является правильным? Должен ли клиент с этого момента прекратить работу?

Это эквивалентно, например, ситуации, когда поставщик популярного приложения для смартфонов решил прекратить поддерживать существующие версии и вынудить пользователей установить у себя последние версии приложения. Возможно, в этом нет необходимости и можно продолжать поддерживать существующие функции.

Если же по какой-то причине действующие бизнес-сценарии больше не могут использоваться в их нынешнем виде, можно подумать о создании некоторой дополнительной, компенсирующей бизнес-логики.

Hypermedia REST и управление версиями

API Hypermedia REST немного упрощают эту задачу. В частности, элементы управления Hypermedia позволяют развивать API путем динамического определения ссылок на ресурсы и действий. Клиенты сервиса REST будут адаптироваться к изменениям в доступе к сервисам, предусмотрительно игнорируя неизвестные функции.

Довольно часто предоставляется возможность контроля версий API. Это означает введение разных операций и ресурсов, таких как `/car-production/v1/cars`, причем именно версия определяет, какую часть API использовать. Однако контроль версий API противоречит идее простых интерфейсов. В частности, поскольку ресурсы REST API представляют собой сущности предметной области, введение нескольких *версий* автомобиля не имеет смысла в терминах бизнеса. Сущность «автомобиль» идентифицируется по ее URI. Изменение URI, чтобы отразить изменения бизнес-функциональности, означало бы изменение сущности «автомобиль».

Иногда требуются несколько различных представлений или версий одной и той же сущности предметной области — например, JSON-преобразования, которые содержат разные наборы свойств. Это достигается через HTTP-интерфейс путем *согласования содержимого* и определения параметров типа контента. Например, различные представления JSON для одного и того же автомобиля могут запрашиваться через типы контента, такие как `application/json;vnd.example.car+v2`, если их поддерживает соответствующий сервис.

Управление интерфейсами — важный вопрос для распределенных систем. Рекомендуется заранее тщательно прорабатывать API с учетом обратной совме-

стимости. Лучше принять дополнительные меры, такие как ввод специальных операций, не позволяющих API нарушать существующую функциональность, чем создать чистый интерфейс, нарушающий работу клиентов.

Документирование границ

Границы приложений, определяемые API для вызова бизнес-логики, должны быть открытыми для клиентов — например, других приложений в системе. Возникает вопрос: какая информация должна быть документирована?

Ограниченный контекст приложения является частью карты контекстов. Таким образом, задачи предметной области должны быть четко описаны. Приложение выполняет определенные бизнес-сценарии в рамках своего контекста.

Эта информация о предметной области должна быть документирована в первую очередь. Клиенты должны знать, какие возможности предоставляет приложение. Это касается бизнес-сценариев, а также передаваемой информации и владения данными.

Задачи приложения для производства автомобилей — это сборка автомобилей в соответствии с точными спецификациями. Приложение владеет информацией о состоянии выпускаемых автомобилей в течение всего процесса сборки, пока автомобиль не дойдет до конца производственной линии и не будет готов к доставке потребителю. Приложение может опрашиваться для предоставления новых данных о состоянии процесса производства автомобиля.

Описание предметной области приложения должно содержать информацию, необходимую клиентам, с точным, но не слишком многословным описанием выполняемых задач, раскрывая только то, что клиенты *должны знать*.

Помимо предметной области, существуют еще технические составляющие, которые следует документировать. Клиентские приложения должны быть запрограммированы в соответствии с системным API. Они требуют информации о протоколах связи и форматах данных.

В главе 2 мы рассмотрели несколько протоколов связи и способы их реализации. На момент написания книги одним из самых распространенных протоколов был HTTP в сочетании с типами содержимого JSON и XML. Предположим, что мы тоже используем HTTP. Что в этом случае нужно документировать?

Конечные точки HTTP, особенно те, которые соответствуют ограничениям REST, представляют сущности предметной области в виде ресурсов, локализуемых по URL-адресам. В первую очередь необходимо задокументировать доступные URL. Клиенты будут подключаться к этим URL для выполнения тех или иных бизнес-сценариев. Например, URL `/car-production/cars/<car-id>` будет ссылаться на конкретный автомобиль, определяемый по его идентификатору.

Необходимо также документировать тип контента и подробную информацию о его преобразовании. Клиенты должны знать структуру и свойства используемого типа контента.

Например, спецификация для создания автомобиля, содержит *идентификатор*, а также сведения о *типе двигателя* и *цвете кузова*. В формате JSON эти данные выглядят следующим образом.

```
{
    "identifier": "<car-identifier>",
    "engine-type": "<engine-type>",
    "color": "<chassis-color>"
}
```

Типы и доступные значения также должны быть задокументированы. Они соответствуют знаниям из предметной области и семантике, описывающей тип двигателя. Важно задокументировать и типы контента, и семантику информации.

В случае с HTTP есть еще ряд документируемых показателей, таких как данные заголовка, которые могут потребоваться, коды состояния, предоставляемые веб-сервисом, и т. д. Разумеется, вся эта документация зависит от задействованной технологии и протокола связи. Предметная область также должна быть частью документации и давать как можно больше сведений.

Документация API приложения является частью программного проекта. Она должна поставляться вместе с приложением соответствующей версии.

Для того чтобы документация всегда соответствовала версии приложения, она должна быть частью репозитория проекта и храниться в системе контроля версий. Поэтому для ее создания настоятельно рекомендуется использовать не двоичные, а текстовые форматы, например документы Word. Хорошо себя зарекомендовали и простые языки разметки, такие как *AsciiDoc* или *Markdown*.

Преимущество сохранения документации непосредственно в проекте, вместе с исходными кодами приложения, заключается в том, что при этом версия документации всегда будет соответствовать разработанному сервису. Инженеры могут изменять их одновременно, что позволит избежать расхождения документации с версией сервиса.

Есть много инструментов, поддерживающих документирование границ приложений в зависимости от технологии связи. Например, для веб-сервисов HTTP широко применяется *спецификация OpenAPI* в сочетании со *Swagger* в качестве фреймворка документации. Swagger выводит определения API в виде читаемого в браузере HTML, что позволяет программистам легко идентифицировать предлагаемые ресурсы и способ их использования.

Сервисы Hypermedia REST позволяют избавиться и от самой большой проблемы сервисной документации. Предоставление информации о том, какие ресурсы доступны по ссылкам, избавляет от необходимости документировать URL-адреса. Фактически сервер получает контроль над тем, как создаются URL. Клиенты указывают только точку входа, например `/car-manufacture/`, и переходят по ссылкам, предоставляемым Hypermedia на основе их отношений. Сервер сам знает, из чего состоит URL-адрес автомобиля, и эта информация явно не документируется.

Это особенно справедливо для элементов управления Hypermedia, которые не только направляют клиентов к ресурсам, но и сообщают о том, как их исполь-

зовать. Так, сервис «производство автомобилей» говорит клиенту, как выполнять действие `create-car`: нужен POST-запрос по адресу `/car-manufacture/cars`, куда входит тело запроса с типом содержимого JSON, который имеет свойства `identifier`, `engine-type` и `color`.

Клиент должен знать семантику всех отношений и имен действий, а также свойства и их происхождение. Это, конечно, логика клиента. Вся информация о том, как использовать API, является частью API. Проектирование служб REST избавляет от необходимости писать обширную документацию.

Последовательность или масштабируемость?

Коммуникация является неотъемлемой частью распределенной системы. Поскольку компьютерные сети нельзя назвать надежными, даже если это внутренняя сеть компании, организация надежной связи является насущной необходимостью. Для правильного поведения бизнес-сценариев необходимо обеспечить надежную коммуникацию.

Ранее в этой книге мы представили так называемую теорему CAP, согласно которой для распределенных хранилищ данных можно гарантировать не более двух из трех известных ограничений. Можно только выбрать, что именно мы хотим гарантировать в системе — согласованность или горизонтальную масштабируемость. Это сильно влияет на обмен данными в распределенной среде.

В общем случае корпоративные системы должны последовательно выполнять бизнес-сценарии. Бизнес-логика должна переводить систему из одного согласованного состояния в другое.

В распределенных системах общее согласованное состояние подразумевает, что бизнес-сценарии, которые обращаются к внешним системам, должны гарантировать, что вызываемая внешняя логика также будет согласованной. Такой подход приводит к распределенным транзакциям. Бизнес-сценарии, вызываемые в системе, включая все внешние системы, выполняются в режиме *«все или ничего»*. Это подразумевает необходимость заблокировать все задействованные распределенные функции до тех пор, пока все распределенные приложения не решат все свои задачи.

Естественно, этот подход не масштабируется. Поскольку система распределенная, управление транзакциями будет происходить по потенциально медленной сети. Получается узкое место, которое может вызвать блокировку, поскольку задействованные приложения должны блокироваться и находиться в состоянии ожидания в течение довольно длительное время.

Вообще говоря, синхронная, согласованная коммуникация рекомендуется только для тех приложений, которые не требуют подключения более чем к двум другим приложениям одновременно. Тесты производительности, а также опыт эксплуатации говорят о том, что выбранный сценарий связи довольно хорошо масштабируется для бизнес-сценария и среды.

Асинхронная связь предпочтительна по причине масштабируемости. Распределенные системы с асинхронной связью по определению не являются

согласованными. Асинхронная связь осуществляется на логическом уровне, где синхронные вызовы только инициируют бизнес-логику, не ожидая согласованного результата.

Далее мы рассмотрим причины и способы построения асинхронной и в итоге согласованной коммуникации в распределенных приложениях.

Регистрация событий, архитектура, управляемая событиями, и CQRS

Корпоративные приложения традиционно создаются с использованием модельного подхода, основанного на атомарном методе *«создание, чтение, обновление, удаление» (create, read, update, delete, CRUD)*.

Текущее состояние системы, включая состояние сущностей предметной области, отражается в реляционной базе данных. Если сущность предметной области обновляется, то ее новое состояние, включая все свойства, помещается в базу данных, а старое состояние удаляется.

Концепция CRUD требует, чтобы приложения сохраняли согласованность. Для того чтобы гарантировать правильное отражение состояния сущности предметной области, все вызовы бизнес-сценариев должны выполняться согласованно, с синхронизацией изменений сущностей.

Недостатки CRUD-систем

Эта согласованность является еще одним недостатком CRUD-систем, обычно используемых для создания приложений.

Масштабируемость

Необходимость синхронизации ограничивает возможности масштабирования системы. Все транзакции выполняются с одной сущностью реляционной базы данных, что в итоге создает узкое место при масштабировании системы.

Это в конце концов становится проблемой для систем с очень большими рабочими нагрузками или огромным количеством пользователей. Однако для подавляющего большинства корпоративных приложений масштабируемости реляционных баз данных вполне достаточно.

Конкурирующие транзакции

Еще одна проблема, возникающая при использовании моделей на основе CRUD, — это обработка конкурирующих транзакций. Бизнес-сценарии, включающие в себя одни и те же сущности предметной области и действующие одновременно, должны гарантировать, что итоговое состояние сущностей будет согласованным.

Изменение имени пользователя с одновременным обновлением кредитного лимита его учетной записи не должно приводить к потере обновлений. Реализация должна гарантировать, что общий результат обеих транзакций будет непротиворечивым.

То, что конкурирующие транзакции полагаются на оптимистическую блокировку, обычно вызывает сбой транзакций. Очевидно, такой подход неидеален с точки зрения пользователя, но он по крайней мере обеспечивает согласованность, не подавляет транзакции, и они не теряются бесследно.

Однако такой подход может привести к излишним блокировкам. С точки зрения теории бизнеса необходимо иметь возможность одновременно редактировать имя пользователя и кредитный лимит учетной записи.

Воспроизводимость

Поскольку приложение сохраняет только текущее состояние, вся информация о предыдущих состояниях теряется. Состояние всегда перезаписывается новыми значениями. Поэтому бывает трудно воспроизвести, как приложение пришло в текущее состояние. Если текущее состояние было ошибочно вычислено по исходным вызовам бизнес-сценариев, то впоследствии исправить эту ситуацию нельзя.

Часть сценариев явно требует воспроизводимости по юридическим соображениям. Поэтому некоторые приложения включают в себя журналы аудита, в которые постоянно заносится определенная информация о том, что происходит с системой.

Регистрация событий

Регистрация событий — это концепция, позволяющая решить проблему воспроизводимости и устранить этот недостаток систем на основе CRUD.

Системы с регистрацией событий вычисляют текущее состояние системы по атомарным событиям, которые произошли в прошлом. События представляют собой отдельные вызовы бизнес-сценариев, включающие в себя информацию, содержащуюся в вызовах.

Текущее состояние не хранится постоянно, но его можно восстановить, воспроизводя все события одно за другим. Сами события происходили в прошлом и неизменны.

Рассмотрим пример: пользователь со своими характеристиками вычисляется на основе всех связанных с ним событий. Применяя одно за другим события `UserCreated`, `UserApproved` и `UserNameChanged`, мы создадим текущее представление пользователя по его последним событиям (рис. 8.2).

События содержат самодостаточную информацию, в основном касающуюся соответствующего бизнес-сценария. Например, событие `UserNameChanged` содержит метку времени и измененное имя пользователя — и никакой другой информации о нем. Таким образом, информация события является атомарной.

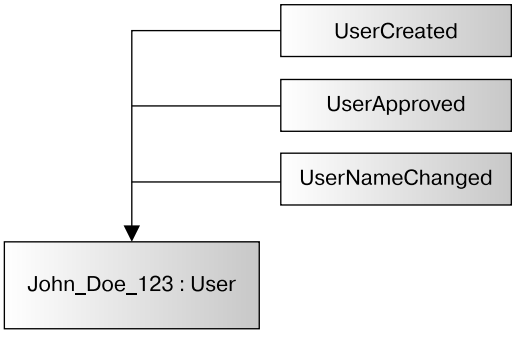

Рис. 8.2

События никогда не изменяются и не удаляются. Если сущность предметной области будет удалена из приложения, произойдет соответствующее событие удаления, например `UserDeleted`. После применения всех этих событий текущее состояние системы больше не будет содержать этого пользователя.

Преимущества

В приложении с регистрацией событий вся информация о приложении хранится в виде атомарных событий. Это все сведения о том, как приложение оказалось в текущем состоянии. Для того чтобы воспроизвести текущее состояние с целью отладки, можно рассмотреть все события и каждое изменение, вносимое ими в систему.

То, что все, что произошло с системой, хранится в виде атомарных событий, имеет ряд преимуществ не только с точки зрения отладки. Эту информацию можно использовать для воспроизведения происходившего с системой при проведении системных тестов в среде эксплуатации. Затем тесты могут повторять точные вызовы бизнес-сценариев, которые были сделаны ранее — в процессе эксплуатации. Это важное преимущество, особенно для системных тестов и тестов производительности.

Это верно и для статистики, в которой атомная информация используется для сбора данных о применении приложения. Это позволяет задействовать бизнес-сценарии и данные, полученные после развертывания приложения.

Предположим, что менеджер хочет узнать, сколько пользователей было создано в понедельник, после того как приложение проработало в течение двух лет. В CRUD-системе эту информацию пришлось бы явно сохранять до момента вызова бизнес-сценария. Если в прошлом бизнес-сценарий явно не запрашивался, то он может быть добавлен только как новая функция и в будущем сможет создавать новые значения.

При регистрации событий такая функциональность становится возможной. Поскольку информация о том, что произошло с системой, сохраняется, бизнес-сценарии, которые будут создаваться в дальнейшем, смогут работать с ранее созданными данными.

Разумеется, эти выгоды можно получить и без использования распределенных систем. Монолитное независимое приложение может быть основано на модели с регистрацией событий и пользоваться теми же преимуществами.

Согласованность в реальном мире

Прежде чем углубиться в изучение согласованности и масштабируемости распределенных систем, рассмотрим пример того, насколько согласованным является реальный мир. Корпоративные приложения обычно построены так, чтобы по возможность обеспечить полную согласованность. Однако реальный мир представляет собой очень распределенную и несогласованную систему.

Представьте, что вы голодны и хотите съесть гамбургер. Вы идете в ресторан, садитесь за столик и просите официанта принести гамбургер. Официант примет заказ. Но, хотя заказ и принят, это вовсе означает, что вы получите еду. Процесс заказа не является полностью согласованным.

В этот момент многое может пойти не так. Например, повар может сказать официанту, что, к сожалению, только что использовал последнюю булочку для гамбургеров и сегодня их больше не будет. Таким образом, несмотря на то что ваш заказ был транзакционно принят, официант вернется и скажет, что выполнить его невозможно.

Теперь вместо того, чтобы попросить вас уйти, официант может предложить другое блюдо. И если вы голодны и замена вас устраивает, то можете в конце концов получить еду.

Вот каким образом высокораспределенный реальный мир обрабатывает транзакции бизнес-сценариев.

Если создать полностью согласованную модель ресторана, то сценарий выглядел бы иначе. Для того чтобы гарантировать, что заказ будет принят только в том случае, если есть возможность приготовить еду, весь ресторан должен быть закрыт. Клиентам придется подождать и прекратить разговоры, пока официант идет на кухню и заказывает еду повару. Поскольку и после заказа многие процессы могут пойти неправильно, вся транзакция заказа фактически должна блокироваться до тех пор, пока еда не будет полностью приготовлена.

Очевидно, такой подход нам не подходит. Ведь реальный мир построен на сотрудничестве, намерении выполнить обещанное или, если сделать это окажется невозможно, по крайней мере устранить проблемы.

Это означает, что реальный мир в итоге действует согласованно. В конечном счете система ресторана окажется в согласованном состоянии, но не обязательно будет находиться в нем в любой момент. На практике это означает, что первоначально принятые заказы в действительности могут оказаться невыполнимыми.

Реальные процессы представлены в виде намерений, или *команд*, таких как заказ гамбургера, и атомарных результатов, или событий, таких как то, что заказ принят. Затем события вызывают новые команды, которые приведут к новым результатам или сбоям.

Архитектуры с регистрацией событий

Теперь вернемся к теме распределенных систем. Как и в случае с рестораном, распределенные системы, которые обмениваются данными согласованным образом, посредством распределенных транзакций, не могут быть масштабированы.

В архитектурах с регистрацией событий эта проблема решена. Связь в них осуществляется посредством асинхронных событий, которые публикуются и потребляются надежным образом.

Таким образом, каждая согласованная транзакция бизнес-сценариев разделяется на несколько меньших согласованных транзакций. Это приводит к тому, что в итоге весь бизнес-сценарий будет согласованным.

Рассмотрим пример того, как бизнес-сценарий заказа гамбургера может быть реализован в архитектуре с регистрацией событий (рис. 8.3). Система ресторана состоит не менее чем из двух распределенных приложений — «официант» и «повар». Приложения ресторана коммуницируют, слушая события друг друга. Приложение «клиент» будет связываться с «официантом», чтобы запустить бизнес-сценарий.

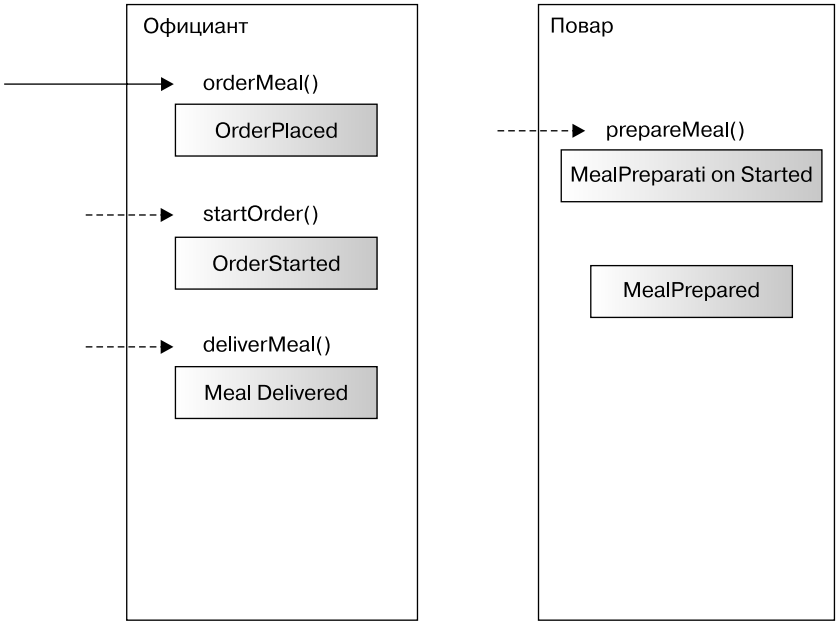

Рис. 8.3

«Клиент» заказывает еду у приложения «официант», что приводит к появлению события `OrderPlaced`. После того как событие было надежно опубликовано, метод `orderMeal()` возвращает вызов. Таким образом, «клиент» может параллельно делать другую работу.

Система «повар» принимает событие `OrderPlaced` и проверяет, можно ли выполнить заказ, используя доступные в настоящее время ингредиенты. Если сделать это невозможно, то «повар» генерирует другое событие, например `OrderFailedInsufficientIngredients`, и тогда «официант» изменит состояние заказа на «не выполнен».

Если приготовление еды успешно начато, «официант» получает событие `MealPreparationStarted` и обновляет состояние заказа на `OrderStarted`. Если «клиент» в это время спросит у «официанта» о состоянии своего заказа, тот сможет ответить соответствующим образом.

Когда приготовление еды закончится, появится событие `MealPrepared`, которое сообщит «официанту», что пора доставить заказ.

Итоговая согласованность в архитектурах с регистрацией событий

Бизнес-сценарий заказа еды является в итоге согласованным. Надежная публикация событий по-прежнему гарантирует, что все клиенты в итоге узнают о состоянии своих заказов. Это неплохо, если обработка заказа не начинается немедленно или по какой-то причине выполнить его не удастся. Однако не должно случиться так, чтобы заказ потерялся в системе из-за того, что приложение было недоступно. Это необходимо гарантировать при публикации событий.

Здесь также используются транзакции, но в гораздо меньших масштабах и не связанные с внешними системами. Это позволяет распределенным системам реализовывать транзакционные бизнес-сценарии, сохраняя возможность горизонтального масштабирования.

Для таких концепций, как архитектура с регистрацией событий, требуется высокая надежность. Это следует учитывать при разработке решений и выборе технологий.

Введение в CQRS

Теперь соберем воедино причины выбора архитектуры с регистрацией событий и собственно регистрации событий.

В архитектурах с регистрацией событий связь осуществляется посредством атомарных событий. Имеет смысл применять комбинированный подход и строить систему с регистрацией событий, используя события как источник истины о системе. Таким образом преимущества обоих методов будут сочетаться, что позволит создавать масштабируемые по горизонтали системы с регистрацией событий.

Но как моделировать приложения с регистрацией событий, в которых модель предметной области основана на событиях? И как эффективно вычислять и возвращать текущее состояние сущностей предметной области?

Моделирование таких приложений описывается принципом *разделения задач на команды и запросы (Command Query Responsibility Segregation, CQRS)*. Он вытекает из архитектуры с регистрацией событий и основан на ней.

Основные принципы

Как следует из названия, в CQRS разделены задачи команд и запросов, а именно чтение и запись.

Команда изменяет состояние системы, в конечном счете производя событие. Она не может возвращать данные. Команда либо завершается успешно и возвращает ноль или производит новые события, либо завершается неудачно и возвращает сообщение об ошибке. События генерируются надежно.

Запрос извлекает и возвращает данные без побочных эффектов в системе. Он не может изменить состояние.

На языке Java команда действует как метод, например `void doSomething()`, который изменяет состояние. Запрос действует как метод чтения `String getSomething()`, не влияющий на состояние системы. Эти принципы кажутся простыми, но имеют определенное влияние на архитектуру системы.

Обязанности команд и запросов делятся на несколько задач, позволяя использовать приложения CQRS в полностью независимых приложениях, которые выполняют либо запись, либо чтение. Как же их разрабатывать и реализовывать этот подход?

Разработка

В соответствии с принципами архитектуры с регистрацией событий системы чтения и записи взаимодействуют исключительно через события. Последние распространяются через хранилище или концентратор событий. Никакой другой связи, кроме систем записи, создающих события, и систем чтения и записи, потребляющих события и обновляющих свое внутреннее состояние, не существует.

На рис. 8.4 показана архитектура CQRS-системы.

Сервисы команд и запросов потребляют события из хранилища. Это единственный способ коммуникации между ними.

Все сервисы поддерживают представление текущего состояния, которое отражает состояние сущностей предметной области. Такими сущностями могут быть, например, *заказы еды* или *автомобили*, причем фиксируется последнее состояние их свойств. Это состояние сохраняется в памяти или записывается в базу данных.

Эти представления позволяют сохранить текущее состояние в системе. Истиной в последней инстанции являются атомарные события, записанные в хранилище событий. Каждый экземпляр приложения самостоятельно обновляет свои представления состояний, потребляя события из хранилища и применяя их.

Командные сервисы содержат бизнес-логику, инициирующую изменения в системах. Они производят события, которые заносят в хранилище после возможной проверки команды, используя их представления состояний.

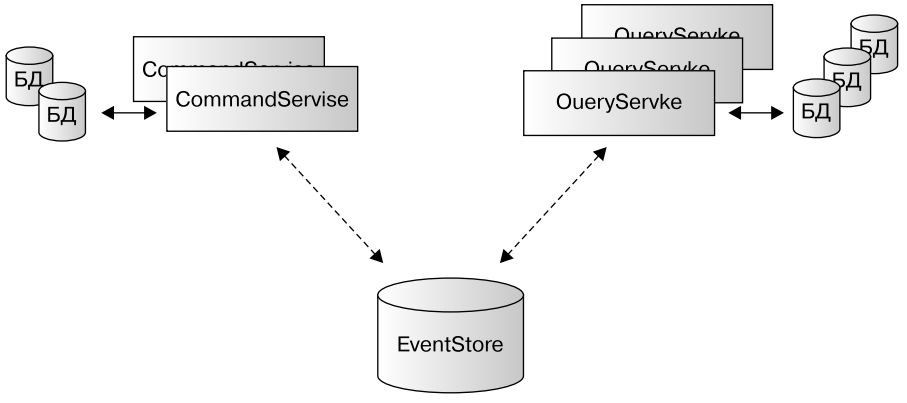

Рис. 8.4

Для того чтобы лучше понять, как движется информация, рассмотрим этот процесс на примере заказа еды (рис. 8.5).

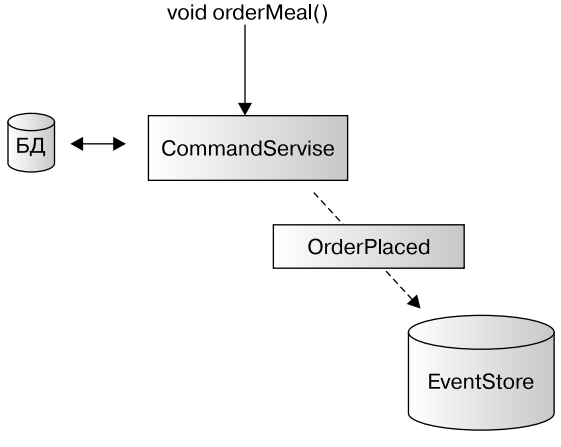

Рис. 8.5

Клиент заказывает еду в экземпляре сервиса команд. После возможной проверки представления сервис команд создает событие `OrderPlaced` и заносит его в хранилище событий. Если публикация события прошла успешно, то возвращается метод `orderMeal()`. Клиент может продолжить работу.

Сервис команд может создать идентификатор заказанной еды, который впоследствии может использоваться, например, как универсальный уникальный идентификатор (рис. 8.6).

Хранилище событий делает событие доступным для всех потребителей, которые соответственно обновляют свои внутренние представления. Клиент может получить доступ к состоянию заказанной еды в сервисе запроса по его идентификатору. В ответ сервис запроса будет выдавать последние представления заказа.

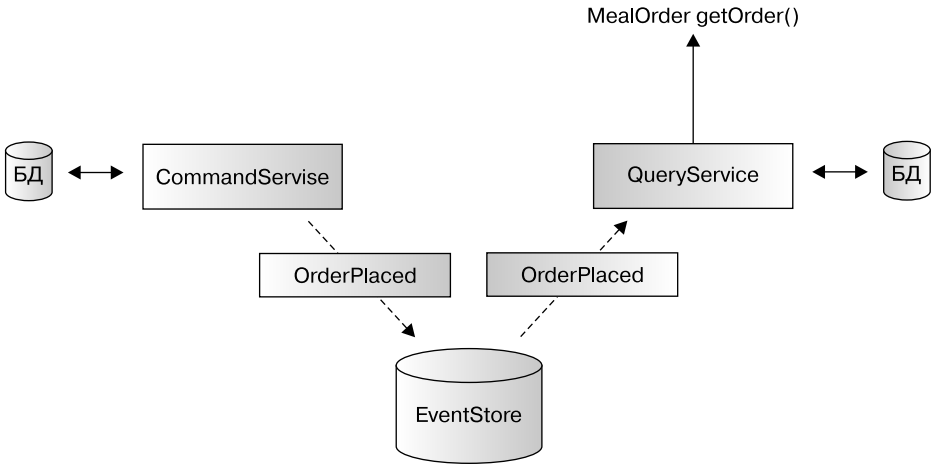

Рис. 8.6

Для того чтобы продолжить обработку заказа, административный центр может вызывать последующие команды, которые также обрабатывают событие (рис. 8.7).

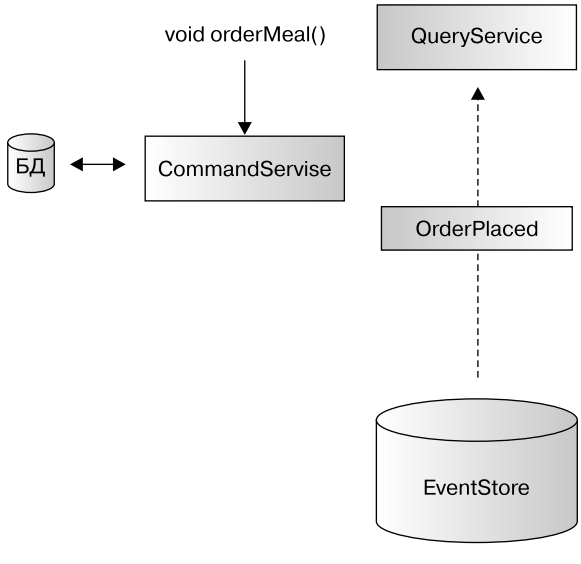

Рис. 8.7

Обработчик событий будет ожидать события `OrderPlaced` и после его получения вызывать бизнес-сценарий системы «повар» `prepareMeal()`. Эта команда может привести к появлению новых событий.

О том, как реализовать CQRS, вы узнаете в разделе «Реализация микросервисов в Java EE».

Преимущества

CQRS позволяет масштабировать распределенные приложения не только по горизонтали, но и независимо от задач записи и чтения. Например, количество реплик сервиса запросов может отличаться от количества сервисов команд.

Нагрузка чтения и записи в корпоративных приложениях обычно распределяется неравномерно: операций чтения значительно больше, чем операций записи. В этих случаях количество экземпляров чтения может быть уменьшено независимо от экземпляров записи. В системе на основе CRUD это было бы невозможно.

Другое преимущество заключается в том, что каждый сервис может соответствующим образом оптимизировать свое представление состояния. Например, постоянное сохранение сущностей предметной области в реляционной базе данных не всегда является лучшим решением. Можно хранить представления в памяти и пересчитывать все события при запуске приложения. Дело в том, что экземпляры чтения и записи могут свободно выбирать и оптимизировать свои представления в зависимости от обстоятельств.

Побочным эффектом этого подхода является также то, что CQRS обеспечивает возможность обработки отказов на стороне чтения. Если хранилище событий недоступно, то никакие новые события не могут быть опубликованы, поэтому в системе не могут быть вызваны бизнес-сценарии, изменяющие ее состояние. В системах на основе CRUD это соответствует отсутствию доступа к базе данных. Однако в CQRS-системах сервисы запросов могут по крайней мере передавать последнее состояние из своих представлений.

Представления состояний в CQRS-системах решают проблему масштабируемости систем с регистрацией событий. Такие системы вычисляют текущее состояние приложения по атомарным событиям. Со временем по мере поступления новых событий при каждом вызове операции этот процесс выполняется все медленнее. Представления сервисов команд и запросов устраняют эту необходимость, постоянно применяя последние события.

Недостатки

У CQRS-систем есть не только преимущества, но и недостатки. Вероятно, одним из самых больших является то, что большинство программистов не знакомы с их концепцией, структурой и реализацией. Это может вызвать трудности, если данный подход будет выбран для корпоративного проекта. В отличие от систем на основе CRUD CQRS потребует дополнительной подготовки и знания ноу-хау.

Как в любых распределенных системах, в CQRS-системах имеется больше приложений, чем при использовании CRUD. Как уже говорилось, и для распределенных систем в целом, и для данного случая это требует дополнительных усилий. Кроме того, нужно будет хранилище событий.

В отличие от схем, показанных на рисунках, здесь не обязательно иметь команды и запросы в виде двух или более независимых приложений. Пока функции взаимодействуют только через события, опубликованные хранилищем событий, чтение и запись может выполнять одно приложение. Это может привести,

в частности, к созданию одного приложения для официанта и повара, которое по-прежнему можно горизонтально масштабировать. Это хороший компромисс, так как не требуется индивидуальное масштабирование сервисов чтения и записи.

Коммуникация

Создание CQRS-систем — это один из способов реализации асинхронной согласованной в итоге коммуникации. Как мы уже знаем, существует множество форм коммуникации, как синхронных, так и асинхронных.

Для того чтобы приложения были масштабируемыми, распределенные системы не должны опираться на синхронную связь, в которой задействованы несколько систем. Это приводит к распределенным транзакциям.

Один из способов реализации масштабируемости при использовании независимых от технологии синхронных протоколов связи — это моделирование логически асинхронных процессов. Например, такие протоколы связи, как HTTP, могут применяться для запуска обработки, которая выполняется асинхронно, а вызывающий абонент сразу возвращает значение. Это обеспечивает итоговую согласованность, но позволяет масштабировать систему.

Также требуется решить, различают ли приложения, образующие распределенную систему, внутреннюю и внешнюю связь. В CQRS этот подход реализован так: клиентам предлагаются внешние интерфейсы, например HTTP, для клиентов, а сами сервисы общаются через хранилище событий. Моделирование асинхронных процессов с обращением к ним через единый протокол в данном случае не различается.

В целом при планировании распределенных систем рекомендуется отдавать предпочтение доступности, то есть масштабируемости, а не согласованности. Есть много вариантов, сочетающих асинхронную связь с регистрацией событий, и CQRS — лишь один из них.

В следующем разделе мы обсудим необходимость самодостаточных приложений.

Архитектуры микросервисов

Мы рассмотрели причины, задачи и преимущества распределенных систем, а также некоторые способы связи и обеспечения согласованности. Теперь обратим внимание на архитектуру распределенных приложений.

Совместный доступ к данным и технологиям в корпоративных системах

Общей идеей для корпоративных систем является совместное многократное применение технологий, а также основных данных. Ранее мы рассматривали совместное использование модулей Java и недостатки этого подхода. А как обстоит

дело с совместным использованием технологий и моделей данных в распределенных системах?

Несколько приложений, образующих корпоративную систему, часто реализуются на базе одних и тех же технологий. Это естественно для приложений, создаваемых одной командой или несколькими тесно взаимодействующими командами. Поэтому очень часто возникает мысль о совместном использовании технологий в приложениях.

Проекты могут задействовать основные модули, чтобы исключить дублирование в системе. Обычный способ реализации этой концепции — общие модели. В организации может быть только один модуль, который многократно задействуется во всех проектах.

Совместное использование моделей приводит к вопросу о многократном применении сохраненных сущностей предметной области и объектов переноса. Если сущность предметной области была сохранена в базе данных, ее же можно потом явно оттуда извлечь, не так ли?

Распространенные базы данных полностью противоречат концепции распределенных систем. Они тесно взаимодействуют с приложениями. Изменение схемы или технологии тесно связывает жизненный цикл приложения и проекта. Экземпляры распространенных баз данных мешают масштабированию приложений, что делает бессмысленным применение распределенных систем.

То же самое верно и для совместного использования технологий в целом. Как было показано в предыдущих главах, обычно применяемые модули и зависимости вводят технические ограничения в реализации. Они связывают приложения и ограничивают их независимость в смысле изменений и жизненного цикла. Командам разработчиков приходится тесно общаться и обсуждать изменения, даже если те не повлияют на границы использования приложения.

Если рассматривать знания и задачи предметной области в рамках одной системы, то совместное использование данных и технологий довольно бессмысленно. По-настоящему эффективное совместное использование технологий возможно между системами.

Но задача заключается в том, чтобы внедрять приложения, зависящие только от их назначения, с одной стороны, и документированных протоколов связи — с другой. Поэтому рекомендуется выбирать независимость, пускай и с потенциальным дублированием, а не связывание технологий.

Совместное использование ресурсов, за исключением точек соприкосновения в контекстной карте системы, должно служить предупреждением для инженеров. В описании задач разных приложений следует ясно показать, что совместно используемые модели и данные принадлежат разным контекстам. Каждое приложение должно само отвечать за решение своих задач.

Архитектуры без разделения ресурсов

Учитывая эти соображения, рекомендуется создавать приложения без совместно используемых технологий или данных. Каждое приложение находится в своих границах и выполняет свои задачи, обмениваясь данными согласно контракту.

Архитектуры без разделения ресурсов не зависят от технологий, применяемых библиотек, их данных и схем. Они позволяют свободно выбирать реализации и технологии сохранения данных.

Перевод приложения в распределенной системе с Python на Java не должен влиять на другие приложения, если сохраняется контракт его HTTP-интерфейса.

Если данные задействуются другими приложениями, необходимо явно указывать это в карте контекстов и требовать, чтобы приложение отображало данные через интерфейсы бизнес-логики. Базы данных не используются.

Архитектуры без разделения ресурсов позволяют создавать приложения с независимыми жизненными циклами, определяемыми только явно описанными контрактами. Команды разработчиков свободны в выборе технологий и жизненных циклов проекта. Технология, а также данные, включая базы данных, принадлежат только приложению.

Независимые системы

Архитектуры без разделения ресурсов рано или поздно должны начать взаимодействовать с другими приложениями. Фиксированные контракты должны быть выполнены, документированы и переданы.

Дело в том, что архитектура без разделения ресурсов зависит только от фиксированных контрактов и обязанностей. В случае изменения бизнес-логики контракты должны быть переопределены и переданы заново. За реализацию контрактов отвечает только команда разработчиков приложения.

Взаимозависимые системы состоят из нескольких приложений без разделения ресурсов с явно описанными интерфейсами. Используемые интерфейсы должны быть независимыми от технологии, чтобы не накладывать ограничения на реализацию.

Именно этот принцип лежит в основе микросервисных архитектур. Микросервисы состоят из нескольких взаимозависимых приложений, каждое из которых реализует свою бизнес-логику, а все они, вместе взятые, решают общую проблему.

Название микросервиса не всегда говорит о размере приложения. Приложение должно создаваться одной группой разработчиков. По организационным причинам размер группы не должен быть слишком большим. В Amazon часто отмечают, что команда разработчиков должна быть такой, чтобы ее можно было досыта накормить двумя пиццами.

Перед построением микросервисов нужно рассмотреть причины, по которым создается распределенная система. Если особой необходимости в такой системе нет, создавать ее не стоит. Предпочтение следует отдать монолитным приложениям с разумным набором решаемых задач.

Обычно при проектировании микросервисных архитектур стремятся разделить монолитные приложения со слишком широким кругом решаемых задач, или создаваемые несколькими командами разработчиков, или имеющие разный жизненный цикл. Это можно сравнить с реструктуризацией. Обычно выполнить

реструктуризацию класса, ставшего слишком большим, и разделить его на несколько делегатов проще, чем сразу написать идеальный сценарий.

В целом, всегда рекомендуется строить систему с учетом бизнес-требований и контекстной карты системы, где определены команды разработчиков и жизненные циклы приложений.

Облачные и двенадцатифакторные приложения

В главе 5 были описаны принципы разработки 12-факторных и облачных приложений. Они основательно поддерживают микросервисную архитектуру.

В частности, концепция, основанная на принципе «ничего общего» с наличием взаимозависимых распределенных приложений, хорошо реализуется с использованием принципов контейнерных масштабируемых корпоративных приложений без сохранения состояния.

Принципы 12-факторных приложений и эффективный характер облачных и контейнерных сред поощряют программистов к разработке высокоэффективных микросервисов с управляемой нагрузкой. Тем не менее корпоративная система не обязательно должна быть распределенной, чтобы соответствовать 12-факторным или облачным принципам. Эти подходы целесообразно применять и при создании монолитных приложений.

Когда микросервисы нужны, а когда — нет

В последние годы в индустрии программного обеспечения поднялась настоящая шумиха вокруг архитектуры микросервисов. Как всегда бывает в таких случаях, инженерам следует спросить себя: что стоит за очередными модными словами и имеет ли смысл это внедрять? Всегда полезно изучить новую технологию и ее методы, но вовсе не обязательно сразу же использовать.

Причины применения микросервисов те же, что и у распределенных систем в целом. Есть технические причины — например, приложение требует независимого жизненного цикла развертывания. Есть также причины, диктуемые бизнес-требованиями, ситуацией в командах разработчиков и режимами работы проекта.

Часто причиной использования микросервисных архитектур называют масштабируемость. Как мы уже увидели, изучая архитектуры с регистрацией событий, монолитные приложения нельзя масштабировать до бесконечности. Вопрос лишь в том, является ли масштабируемость истинной проблемой.

Существуют крупные компании, в которых бизнес-логика, реализованная в виде монолитного приложения, обслуживает огромное количество пользователей. Прежде чем рассматривать распределенную среду как решение проблем масштабирования, необходимо собрать информацию и статистические данные о производительности.

Инженерам не следует рассчитывать на то, что микросервисная архитектура решит все проблемы. По результатам семинаров и обсуждения модных тенденций,

решения часто принимают на основе незначительного числа свидетельств в пользу той или иной технологии или даже при их отсутствии. Конечно, микросервисы обеспечивают ряд преимуществ, но за них приходится платить определенную цену в виде затрат времени и приложения усилий. В любом случае за разделением задач между несколькими приложениями должны стоять явные требования и веские причины.

Реализация микросервисов в Java EE

Теперь поговорим о том, как создавать микросервисы на Enterprise Java.

В различных обсуждениях и на конференциях язык Java EE называли слишком тяжеловесным и громоздким для микросервисов. Безусловно, это верно для технологии и принципов J2EE, но Java EE предлагает современные экономные способы разработки корпоративных приложений. Эти аспекты подробно рассмотрены в главах 4 и 5, особенно в отношении современных сред.

Java EE действительно хорошо подходит для написания микросервисных приложений. Контейнерные технологии и управление контейнерами поддерживают эту платформу особенно потому, что в Java EE бизнес-логика отделена от реализации.

Приложения с нулевыми зависимостями

Микросервисы в Java EE построены как идеальные приложения с нулевыми зависимостями.

«Тонкие» WAR-приложения развертываются в современных корпоративных контейнерах, которые в свою очередь сами могут собираться в контейнеры, что сводит к минимуму время развертывания. Артефакты развертывания Java EE должны содержать только указанные зависимости. Если есть обоснованная потребность в добавлении сторонних зависимостей, то они должны быть установлены на сервере приложений. Данный подход упрощается благодаря контейнерным технологиям.

Это также вполне соответствует идее архитектуры без разделения ресурсов. Команда разработчиков отвечает за все технологии приложения и за включение соответствующих библиотек в установочный пакет сервера. Это легко сделать с помощью определений инфраструктуры как кода, таких как Dockerfiles.

Серверы приложений

В соответствии с этой концепцией сервер приложений поставляется в контейнере, содержащем только одно приложение. Принцип *«одно приложение — один сервер приложений»* также соответствует идее архитектуры без разделения ресурсов.

Возникает вопрос: не слишком ли велики накладные расходы от серверов приложений, если каждый экземпляр сервера содержит только одно приложе-

ние? Раньше это требовало значительного объема оперативной памяти и места в хранилище.

В этом смысле современные серверы приложений значительно улучшены. Существуют базовые образы контейнеров с серверами, такие как *TomEE*, занимающие всего 150 Мбайт и даже менее и содержащие весь сервер, включая среду выполнения Java и операционную систему. Потребление памяти также значительно оптимизировано благодаря функции динамической загрузки.

Размеры установочных пакетов корпоративных проектов обычно не являются большой проблемой, если только они не чрезмерно велики. Гораздо важнее размер собранных и поставляемых артефактов. Артефакт приложения, который для некоторых технологий может содержать зависимости объемом в несколько мегабайт, собирается и передается многократно, в то время как среда выполнения устанавливается только один раз.

Контейнерные технологии, такие как Docker, используют многоуровневые файловые системы, которые позволяют уменьшать изменяющиеся части. Эта концепция поддерживает приложения с нулевой зависимостью.

То, что каждая сборка при непрерывной поставке будет передавать всего несколько килобайт данных, будет гораздо более целесообразно, чем хранить несколько мегабайт в базовом установочном пакете.

Если размер установочного пакета все равно необходимо сократить, некоторые поставщики приложений позволяют адаптировать контейнер к требуемым стандартам. Особенно это касается инициативы MicroProfile, которая поддерживает нескольких поставщиков серверов приложений и определяет тонкие профили.

Микросервисы Java EE не нужно распространять в виде автономных JAR-файлов. Наоборот, приложения, поставляемые в контейнерах, должны поддерживать применение многоуровневых файловых систем и развертываться в корпоративных контейнерах, находящихся в базовом образе. Отдельные JAR-файлы не удовлетворяют этому принципу.

Есть возможность комбинировать автономные JAR-файлы и «тонкое» развертывание с помощью так называемого полого JAR. Однако при использовании контейнеров это не требуется.

Реализация контуров приложений

Перейдем к реализации контура приложения на Java EE. По сути, это вопрос скорее системной архитектуры, чем реализации.

Связь между микросервисами должна задействовать протоколы, независимые от технологии. Как мы уже знаем, Java EE активно поддерживает HTTP, поскольку и HTTP-сервисы, и REST-сервисы применяют Hypermedia.

В следующем подразделе будет рассмотрена асинхронная связь в системах CQRS посредством публикации и подписки, реализованной с помощью Apache Kafka.

Реализация CQRS

Ранее в этой главе мы рассмотрели причины и концепции, лежащие в основе регистрации событий, архитектур, основанных на управлении событиями, и CQRS. CQRS предлагает интересный подход к созданию распределенных приложений, при котором реализуются масштабируемые в конечном счете согласованные бизнес-сценарии.

На момент написания этой книги технология CQRS вызывала большой интерес, но корпорации мало знали о том, как ее использовать. Появились фреймворки и технологии, направленные на реализацию этого подхода. Однако CQRS — это архитектурный стиль, и для разработки CQRS-систем специализированные фреймворки не нужны.

Подробнее рассмотрим концепцию, которая применяется в Java EE.

Системные интерфейсы

Системные интерфейсы в CQRS используются извне системы для запуска бизнес-сценариев. Например, клиент обращается к системе «официант», чтобы заказать гамбургер.

Эти интерфейсы задействуются извне и идеально реализуются с применением технологически независимых протоколов.

Для REST-подобных HTTP-сервисов это означает, что в командных сервисах реализованы изменяющие ресурсы HTTP-методы, такие как `POST`, `DELETE`, `PATCH` и `PUT`. Сервисы запросов обычно только используют ресурсы с помощью запросов `GET`.

В нашем примере это означает, что клиент делает новый заказа еды в виде `POST`-запроса в ресурс сервиса команд. Аналогично заказы на еду формируются через `GET`-запросы ресурсов через сервисы запросов.

Эти HTTP-интерфейсы относятся к внешней коммуникации. Внутри приложения связь осуществляется через события, которые публикуются с помощью концентратора событий.

Пример сценария в Apache Kafka

В этом примере я буду использовать брокер распределенных сообщений Apache Kafka, так как он обеспечивает высокую производительность и пропускную способность. Это лишь один из многих примеров технологии обмена сообщениями, поддерживающей метод публикации и подписки.

На момент написания книги в Apache Kafka не была полностью реализована семантика JMS. В следующих примерах будет применен зависящий от поставщика клиентский API Kafka.

Согласно концепции публикации и подписки в Apache Kafka сообщения упорядочены по темам. Систему можно конфигурировать таким образом, чтобы обеспечить возможность потребления транзакционных событий в той последовательности, в какой они были созданы, что необходимо для разработки надежных бизнес-сценариев в архитектуре с регистрацией событий.

После распространения брокеры Kafka используют так называемые группы потребителей, чтобы управлять темами и разделами сообщений. Изучение архитектуры Kafka выходит за рамки книги, поэтому при выборе этой технологии рекомендую обратиться к ее документации.

Если коротко, сообщение публикуется в теме и потребляется один раз каждой группой потребителей. Каждая группа состоит из одного или нескольких потребителей. Ее наличие гарантирует, что ровно один потребитель обработает сообщение, которое было опубликовано посредством генератора транзакций.

CQRS-система должна потреблять сообщения в нескольких местах. Приложения, заинтересованные в определенной теме, будут потреблять сообщения и обновлять свои внутренние представления. Поэтому каждое событие будет получено всеми обновляющимися пользователями. Существуют также обработчики событий, которые используют событие для реализации бизнес-логики. Каждое событие темы должен обрабатывать ровно один обработчик событий, иначе процессы будут выполняться несколько раз или не будут выполняться вообще.

Таким образом, концепция групп потребителей в Kafka реализована так, что существует одна группа потребителей обновлений для каждого приложения и одна группа обработчиков событий для каждой темы (рис. 8.8). Это позволяет всем экземплярам получать события, но только один командный сервис обрабатывает бизнес-логику. Таким образом, экземпляры могут масштабироваться, не влияя на общую работу системы.

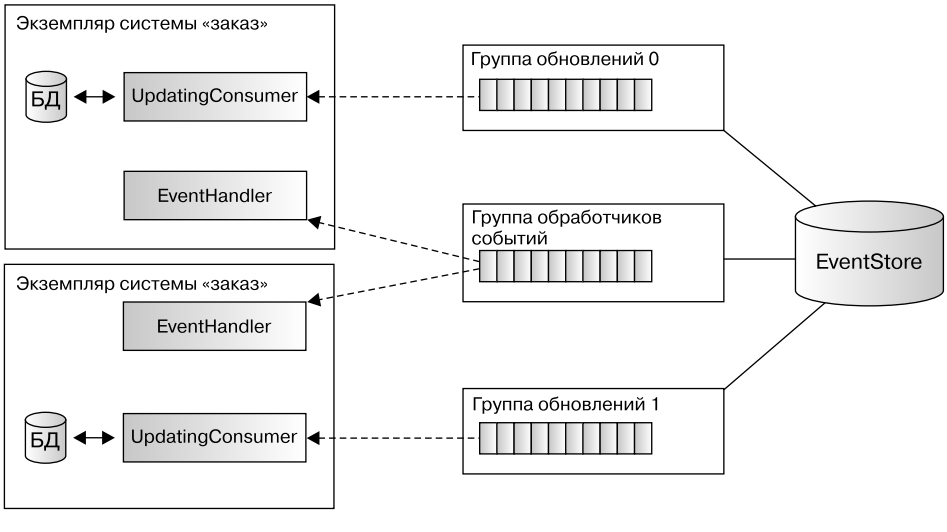

Рис. 8.8

Интеграция Java EE

Для того чтобы интегрировать кластер Apache Kafka в приложение, в этом примере воспользуемся Kafka Java API.

Приложения подключаются к Kafka, чтобы его обновляющиеся потребители и обработчики событий потребляли сообщения. То же самое касается публикации событий.

Используемая технология должна быть инкапсулирована от остальной части приложения. Для того чтобы интегрировать события, программисты могут реализовать функциональность, которая естественным образом подходит для этого сценария, — CDI-события.

CDI-события

События предметной области содержат специфические для событий данные, временную метку и идентификаторы, которые ссылаются на сущность предметной области.

В следующем фрагменте кода приводится пример абстрактного класса `MealEvent` и события `OrderPlaced`:

```
public abstract class MealEvent {

private final Instant instant;

protected MealEvent() {
    instant = Instant.now();
}

protected MealEvent(Instant instant) {
    Objects.requireNonNull(instant);
    this.instant = instant;
}

…
}

public class OrderPlaced extends MealEvent {

    private final OrderInfo orderInfo;

    public OrderPlaced(OrderInfo orderInfo) {
        this.orderInfo = orderInfo;
    }

    public OrderPlaced(OrderInfo orderInfo, Instant instant) {
        super(instant);
        this.orderInfo = orderInfo;
    }

…
}
```

Подобные события предметной области являются ядром приложения. По этим событиям вычисляются представления сущностей предметной области.

Интеграция в Kafka гарантирует, что данные события будут запущены через CDI. За этим наблюдают соответствующие функции, которые обновляют представления состояний или вызывают последующие команды.

Обработчики событий

В следующем фрагменте кода показан обработчик событий системы «повар», вызывающий функции сервиса команд:

```
@Singleton
public class OrderEventHandler {

    @Inject
    MealPreparationService mealService;

    public void handle(@Observes OrderPlaced event) {
        mealService.prepareMeal(event.getOrderInfo());
    }
}
```

Обработчик событий потребляет событие и вызывает контур соответствующего бизнес-сценария приготовления еды. Метод `prepareMeal()` может создавать события, в данном случае `MealPreparationStarted` или `OrderFailedInsufficientIngredients`:

```
public class MealPreparationService {

    @Inject
    EventProducer eventProducer;

    @Inject
    IngredientStore ingredientStore;

    public void prepareMeal(OrderInfo orderInfo) {

        // использовать ingredientStore чтобы проверить наличие продуктов

        if (…)
            eventProducer.publish(new
                OrderFailedInsufficientIngredients());
        else
            eventProducer.publish(new MealPreparationStarted(orderInfo));
}
}
```

Генератор событий надежно публикует события в кластере Kafka. Если публикация заканчивается неудачно, то вся обработка события должна быть остановлена и впоследствии повторена.

Представление состояния

Потребители, обновляющие представление состояния, также потребляют CDI-события. В следующем фрагменте кода показан компонент, который содержит представления состояния заказа еды:

```
@Stateless
public class MealOrders {

    @PersistenceContext
```

```
    EntityManager entityManager;

    public MealOrder get(UUID orderId) {
        return entityManager.find(MealOrder.class, orderId.toString());
    }

    public void apply(@Observes OrderPlaced event) {
        MealOrder order = new MealOrder(event.getOrderInfo());
        entityManager.persist(order);
    }

    public void apply(@Observes OrderStarted event) {
        apply(event.getOrderId(), MealOrder::start);
    }

    public void apply(@Observes MealDelivered event) {
        apply(event.getOrderId(), MealOrder::deliver);
    }

    private void apply(UUID orderId, Consumer<MealOrder> consumer) {
        MealOrder order = entityManager.find(MealOrder.class,
            orderId.toString());
        if (order != null)
            consumer.accept(order);
    }
}
```

В этом простом примере показано состояние заказов на еду в реляционной базе данных. Когда поступает новое CDI-событие, состояние заказов обновляется. Текущее состояние может быть получено с помощью метода `get()`.

Сущность предметной области «заказ еды» сохраняется через JPA. В ней содержится состояние заказа, которое обновляется через наблюдаемые CDI-события:

```
@Entity
@Table("meal_orders")
public class MealOrder {

    @Id
    private String orderId;

    @Embedded
    private MealSpecification specification;

    @Enumerated(EnumType.STRING)
    private OrderState state;

    private MealOrder() {
        // нужно для JPA
    }

    public MealOrder(OrderInfo orderInfo) {
        orderId = orderInfo.getOrderId().toString();
        state = OrderState.PLACED;

        // описание спецификаций
```

```
    }

    public void start() {
        state = OrderState.STARTED;
    }

    public void deliver() {
        state = OrderState.DELIVERED;
    }

…
}
```

Потребление сообщений Kafka

Часть приложения, потребляющая сообщения, инкапсулирует концентратор сообщений от остального приложения. Концентратор интегрируется посредством активизации CDI-событий при поступлении сообщений. Это, безусловно, характерно для API Kafka и должно считаться образцовым решением.

Обновляемый потребитель подключается к определенной теме через свою группу потребителей. Запуск компонента-синглтона гарантирует, что потребитель будет запущен при запуске приложения. Управляемый контейнером сервис-исполнитель запускает потребитель события в своем потоке:

```
@Startup
@Singleton
public class OrderUpdateConsumer {

private EventConsumer eventConsumer;

    @Resource
    ManagedExecutorService mes;

    @Inject
    Properties kafkaProperties;

    @Inject
    Event<MealEvent> events;

    @PostConstruct
    private void init() {
        String orders = kafkaProperties.getProperty("topic.orders");

        eventConsumer = new EventConsumer(kafkaProperties,
                ev -> events.fire(ev), orders);

        mes.execute(eventConsumer);
    }

    @PreDestroy
мpublic void close() {
        eventConsumer.stop();
    }
}
```

Специфичные для приложения свойства Kafka отображаются через CDI-генератор. Он содержит соответствующие группы потребителей.

Фактически потребление выполняет потребитель события:

```
import org.apache.kafka.clients.consumer.KafkaConsumer;
import java.util.function.Consumer;
import static java.util.Arrays.asList;

public class EventConsumer implements Runnable {

    private final KafkaConsumer<String, MealEvent> consumer;
    private final Consumer<MealEvent> eventConsumer;
    private final AtomicBoolean closed = new AtomicBoolean();

    public EventConsumer(Properties kafkaProperties,
            Consumer<MealEvent> eventConsumer, String... topics) {
        this.eventConsumer = eventConsumer;
        consumer = new KafkaConsumer<>(kafkaProperties);
        consumer.subscribe(asList(topics));
    }

    @Override
    public void run() {
        try {
            while (!closed.get()) {
                consume();
            }
        } catch (WakeupException e) {
            // действия по закрытию
        } finally {
                    consumer.close();
        }
    }

    private void consume() {
        ConsumerRecords<String, MealEvent> records =
                consumer.poll(Long.MAX_VALUE);
        for (ConsumerRecord<String, MealEvent> record : records) {
            eventConsumer.accept(record.value());
        }
        consumer.commitSync();
    }

    public void stop() {
        closed.set(true);
        consumer.wakeup();
    }
}
```

Потребление записей Kafka приводит к появлению новых CDI-событий. Для преобразования классов предметной области конфигурируемые свойства используют сериализаторы и десериализаторы JSON.

События, которые запускаются через CDI и успешно потребляются, записываются в Kafka. CDI-события запускаются синхронно, чтобы гарантировать, что все процессы будут надежно завершены до того, как события будут записаны.

Генерация событий Kafka

Генератор событий публикует события предметной области в концентраторе сообщений. Это происходит синхронно, чтобы можно было полагаться на сообщения, находящиеся в системе. После подтверждения передачи возвращается метод `EventProducer#publish`:

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;

@ApplicationScoped
public class EventProducer {

    private Producer<String, MealEvent> producer;
    private String topic;

    @Inject
    Properties kafkaProperties;

    @PostConstruct
    private void init() {
        producer = new KafkaProducer<>(kafkaProperties);
        topic = kafkaProperties.getProperty("topics.order");
        producer.initTransactions();
    }

    public void publish(MealEvent event) {
        ProducerRecord<String, MealEvent> record = new
            ProducerRecord<>(topic, event);
        try {
            producer.beginTransaction();
            producer.send(record);
            producer.commitTransaction();
        } catch (ProducerFencedException e) {
            producer.close();
        } catch (KafkaException e) {
            producer.abortTransaction();
        }
    }

    @PreDestroy
    public void close() {
        producer.close();
    }
}
```

Подробное изучение API генератора Kafka выходит за рамки книги. Однако необходимо обеспечить надежную отправку событий. Эту логику инкапсулирует компонент генератора событий.

Приведенные примеры демонстрируют только одну из возможностей интеграции Kafka.

Как уже отмечалось, *Java EE Connector Architecture (JCA)* — это еще одна возможность интеграции внешних задач в контейнере приложения. На момент написания книги существовали специфические для конкретного поставщика варианты контейнеров для интеграции обмена сообщениями через JCA. Интересной альтернативой являются уже существующие решения для интеграции концентраторов сообщений, такие как Kafka. Тем не менее разработчикам приложений рекомендуется инкапсулировать специфику технологий в виде единых точек ответственности, а в приложении использовать стандартные функции Java EE.

Контуры приложений

Приложения в CQRS-системе обмениваются сообщениями через внутренние события. Извне могут применяться другие протоколы, такие как HTTP.

Функции запроса и команды, например в системе «официант», отображаются через JAX-RS. Командный сервис предлагает функции для размещения заказов на еду. Для публикации полученных событий в нем используется генератор событий:

```
public class OrderService {

    @Inject
    EventProducer eventProducer;

    public void orderMeal(OrderInfo orderInfo) {
        eventProducer.publish(new OrderPlaced(orderInfo));
    }

    void cancelOrder(UUID orderId, String reason) {
        eventProducer.publish(new OrderCancelled(orderId, reason));
    }

    void startOrder(UUID orderId) {
        eventProducer.publish(new OrderStarted(orderId));
    }

    void deliverMeal(UUID orderId) {
        eventProducer.publish(new MealDelivered(orderId));
    }
}
```

Метод `orderMeal()` вызывается конечной точкой HTTP. Другие методы вызываются обработчиком событий системы «официант». Это приводит к появлению новых событий, которые будут доставлены концентратором событий.

Причина заключается в том, чтобы не активизировать события непосредственно и не вызывать внутреннюю функцию, так как приложение находится в распределенной среде. Могут существовать и другие экземпляры «официанта», потребляющие события из концентратора и соответственно обновляющие свое представление.

Сервис команд содержит ресурс JAX-RS, который используется для заказа еды:

```
@Path("orders")
public class OrdersResource {

    Inject
    OrderService orderService;

    @Context
    UriInfo uriInfo;

    @POST
    public Response orderMeal(JsonObject order) {
        OrderInfo orderInfo = createOrderInfo(order);
        orderService.orderMeal(orderInfo);

        URI uri = uriInfo…

        return Response.accepted().header(HttpHeaders.LOCATION,
            uri).build();
    }

    …
}
```

Сервис запросов раскрывает представления заказа еды. Он загружает из базы данных текущее состояние сущностей предметной области, как показано в `MealOrders`. Эту функцию используют ресурсы JAX-RS сервиса запросов.

Если система «официант» поставляется в единственном экземпляре, который содержит и сервисы команд, и сервисы запросов, то эти ресурсы могут быть объединены. Но нужно гарантировать, что сервисы не будут связываться иначе, чем через механизм событий. В следующем фрагменте кода представлена конечная точка сервиса запроса:

```
@Path("orders")
public class OrdersResource {

    @Inject
    MealOrders mealOrders;

    @GET
    @Path("{id}")
    public JsonObject getOrder(@PathParam("id") UUID orderId) {
        MealOrder order = mealOrders.get(orderId);

        if (order == null)
            throw new NotFoundException();

        // создать JSON-ответ
        return Json.createObjectBuilder()…
    }
}
```

Эти примеры не исчерпывающие, они предназначены лишь для того, чтобы дать читателю представление об интеграции концепций CQRS и концентраторов сообщений в приложениях Java EE.

Дальнейшая интеграция концепций CQRS

Одним из преимуществ систем с регистрацией событий является возможность взять полный набор атомарных событий и воспроизвести их, например, в тестовых сценариях. Системные тесты проверяют реальные бизнес-сценарии, которые выполняются в процессе эксплуатации. Также автоматически ведется журнал аудита, поскольку это действие является частью ядра приложения.

Такой подход позволяет также изменять бизнес-функциональность и воспроизводить отдельные события, а затем либо исправлять ошибки, либо корректировать поведение, либо использовать информацию о событиях при создании новых функций. Это позволяет применять к событиям новые функции так, как если бы они с самого начала были частью приложения.

Если в систему «повар» добавить функцию непрерывного вычисления среднего времени приготовления еды, можно повторно доставить все события и заново вычислить все представления. Поэтому содержимое базы данных будет сброшено, а все события будут повторно доставлены обновленному потребителю, что приведет к вычислению и сохранению нового представления. Kafka позволяет явно повторно доставлять события.

Однако события используются только для обновления представлений состояния, а не для запуска новых команд во время повторов, иначе система оказалась бы в несогласованном состоянии. В приведенном примере это реализовано путем создания в Kafka выделенной группы потребителей для обработчиков событий, которая не сбрасывается, чтобы можно было повторно доставлять события обработчикам событий. События повторно получают только обновившиеся потребители, чтобы заново вычислить внутренние представления состояния.

Суть в том, что благодаря регистрации событий CQRS-системы позволяют реализовать гораздо больше бизнес-сценариев. Благодаря возможности захвата и воспроизведения событий, а также содержащимся в них контекстам и информации о ранее совершенных действиях можно строить расширенные сценарии.

Java EE в эпоху распределенных вычислений

Архитектура микросервисов и распределенных систем, естественно, требует коммуникации, в которой участвуют несколько монолитных приложений. В Java EE есть множество способов реализовать коммуникацию, которые зависят от выбранных протоколов и технологий связи.

При разработке коммуникации необходимо учитывать несколько особенностей. Например, внешние приложения, входящие в систему микросервисов, должны находить экземпляры сервисов. Для того чтобы избежать тесной связи между приложениями и конфигурацией, поисковые сервисы следует сделать

динамическими, вместо того чтобы указывать в конфигурации статические адреса узлов или IP-адреса.

Присущий облачным средам принцип устойчивости касается и коммуникации. Поскольку сеть в любой момент может дать сбой, работоспособность приложения не должна зависеть от того, что скорость соединения может упасть, а связь — вообще оборваться. Приложение должно быть защищено от возможного проникновения в него ошибок.

Обнаружение сервисов

Обнаруживать сервисы можно по-разному: как DNS-поиском, так и реализацией более сложных сценариев, где поиск является частью бизнес-логики и выбирает различные конечные точки в зависимости от ситуации. Он обычно инкапсулирует адресацию внешних систем, отделяя ее от задач приложения. В идеале логика приложения знает только имя логического сервиса, с которым ему нужно взаимодействовать, а фактический поиск происходит вне приложения.

Возможности разработчиков корпоративного приложения зависят от используемой среды выполнения. Контейнерные технологии предлагают функциональность для связывания сервисов по именам, избавляя программистов от лишней работы, а приложение — от решения этих задач. Клиенты соединяются по ссылкам и именам сервисов, таким как имена узлов, которые распознаются технологиями контейнера.

Этот метод работает как в контейнерах Docker, так и во фреймворках управления контейнерами, таких как Docker Compose, Kubernetes и OpenShift. Вся коммуникация осуществляется только по логическим именам сервисов и портов, по которым устанавливается соединение. Это также соответствует принципам 12-факторных приложений.

Поскольку поиск происходит в среде выполнения, приложения будут указывать только имена сервисов. Это справедливо для всей внешней связи, такой как HTTP-соединения, обращения к базе данных или концентратору сообщений. Примеры такой связи приведены в главе 5.

Устойчивая связь

Связь по сети ненадежна — есть множество причин, по которым она может прерваться. Соединения могут осуществляться с задержками, сервисы могут оказаться недоступными, реагировать медленно или доставлять неожиданные ответы.

Для того чтобы ошибки не попадали в приложение, связь должна быть устойчивой.

Проверка ответов

Прежде всего это означает проверку и обработку ошибок на стороне клиента. Независимо от используемой коммуникационной технологии приложения не могут полагаться на то, что внешние системы дадут правильные, неискаженные ответы.

Это не означает, что клиенты должны немедленно отклонить все ответы, если в понимании приложения они неидеальны. Ответы, которые содержат больше информации, чем надо, или несколько отличаются от ожидаемого формата, но по-прежнему понятны, не должны вызывать немедленное появление сообщений об ошибках. Придерживаясь принципа *«быть консервативным в том, что вы делаете, и либеральным в том, что принимаете»*, следует принимать сообщения, если они содержат достаточно информации, чтобы приложение могло выполнить свою работу. Например, дополнительные неизвестные свойства в ответах JSON не должны приводить к отказу в преобразовании объекта.

Разрыв связи и предохранители

Клиенты, выполняющие синхронные вызовы внешних систем, блокируют свое выполнение до тех пор, пока внешняя система не ответит. Если вызов завершится неудачей, это замедлит выполнение приложения или даже приведет к его сбою. Этот факт очень важно учитывать при реализации клиентов.

Прежде всего, как было показано в главе 3, клиентские соединения всегда должны устанавливать разумное время задержки. Задержки не позволяют приложениям попадать в тупиковую ситуацию.

Как уже было показано, для того чтобы динамические исключения не проникали в бизнес-логику, можно использовать перехватчики Java EE.

Этот подход реализован в так называемых *предохранителях*, предназначенных для предотвращения каскадных отказов. Предохранители защищают клиентские вызовы, определяя пороговые значения ошибок и задержек и не допуская дальнейших вызовов в случае сбоя. Применение предохранителей основано на электротехнической модели, где устанавливаемые в зданиях предохранители прерывают замыкание электрической цепи, чтобы предотвратить дальнейший ущерб.

Предохранители на стороне клиента аналогичным образом разрывают связь, предотвращая дальнейшие вызовы, чтобы не повредить приложение или внешнюю систему. Обычно они допускают ошибки и задержки, не превышающие определенного предела, а затем разрывают соединение на определенное время или до тех пор, пока оно не будет снова установлено вручную.

В приложениях Java EE можно реализовать предохранители в виде перехватчиков. Они могут добавить сложную логику, определяющую, когда и как открывать и закрывать соединения, — например, измеряя количество сбоев и задержек.

Далее приведен один из возможных вариантов реализации предохранителя, написанный на псевдокоде. Поведение перехватчика аннотируется для клиентского метода, как в примерах клиентских перехватчиков, продемонстрированных ранее в этой книге:

```
@Interceptor
public class CircuitBreaker {

    …

    @AroundInvoke
```

```
    public Object aroundInvoke(InvocationContext context) {

        // закрыть соединение по истечении времени восстановления

        if (circuit.isOpen())
            return null;

        try {
            return context.proceed();
        } catch (Exception e) {

            // записать исключение
            // увеличить список сбоев
            // если количество сбоев превышает порог, то разорвать соединение

            return null;
        }
    }
}
```

Предохранитель также может измерять время работы сервиса и разрывать соединение, если сервис становится слишком медленным или возникают задержки HTTP-клиента. Для этого предусмотрено несколько библиотек Java EE с открытым исходным кодом, таких как *Breakr* от Адама Бина (Adam Bien), эксперта по Java EE. Момент установки и разрыва соединения, а также необходимость сторонних зависимостей определяется техническими требованиями и сложностью логики.

Для того чтобы создавать приложения с нулевой зависимостью, библиотеки, если в них есть необходимость, должны устанавливаться в контейнер, а не поставляться с артефактами приложения.

Переборки

На кораблях есть переборки, которые делят их корпус на несколько отсеков. Если судно получит пробоину, вода заполнит только один отсек и оно, скорее всего, останется на плаву.

Шаблон *переборки* применяет эту идею к корпоративным приложениям. Если какой-либо из компонентов приложения даст сбой или окажется на пределе нагрузки, то остальная часть приложения все равно сможет выполнять свою задачу. Такая возможность, разумеется, сильно зависит от бизнес-сценария.

Одним из примеров является разделение потоков реализации бизнес-процессов и конечных точек HTTP. Серверы приложений управляют общим концентратором потоков запросов. Если, например, один бизнес-компонент выйдет из строя и заблокирует все входящие запросы, то все доступные потоки запросов в итоге окажутся заняты. В результате из-за отсутствия доступных потоков запросов нельзя будет вызвать другие бизнес-сценарии. Это может случиться, если клиент, в котором неправильно заданы задержки, пытается соединиться с отключенной системой и выполнение блокируется.

Для решения этой проблемы можно использовать асинхронные ресурсы JAX-RS в сочетании с выделенными сервисами управляемых исполнителей. Как уже было показано в этой книге, ресурсы JAX-RS позволяют вызывать бизнес-функции в отдельных потоках, управляемых контейнерами, а не в режиме общего выполнения с применением потока запросов. Разные компоненты могут задействовать независимые концентраторы потоков, что предотвращает распространение сбоев.

Поскольку за управление потоками отвечает сервер приложений, этот подход должен быть реализован в соответствии со стандартами Java EE. Суть его в том, чтобы создать выделенные сервисы-исполнители, которые можно внедрять в нужный момент.

Этот принцип был реализован в разработанной Адамом Бином библиотеке *Porcupine* с открытым исходным кодом, предназначенной для создания выделенных сервисов-исполнителей, использующих `ManagedThreadFactory` для определения концентраторов потоков, в которых контейнеры управляют потоками. Выделенные сервисы-исполнители могут быть конфигурированы и настроены соответствующим образом.

В следующем фрагменте кода показан один из вариантов реализации шаблона переборки, объединяющий в себе асинхронные ресурсы JAX-RS с выделенными сервисами-исполнителями:

```
import com.airhacks.porcupine.execution.boundary.Dedicated;
import java.util.concurrent.ExecutorService;

@Path("users")
@Produces(MediaType.APPLICATION_JSON)
public class UsersResource {

    @Inject
    @Dedicated("custom-name")
    ExecutorService executor;

    @GET
    public CompletionStage<Response> get() {
        return CompletableFuture
                .supplyAsync(this::getUsers, executor)
                .thenApply(s -> Response.ok(s).build());
    }

    …
}
```

Бизнес-сценарий выполняется в управляемом потоке, предоставляемом сервисом-исполнителем, чтобы позволить потоку запроса возвращаться и обрабатывать другие запросы. Это дает другим функциям приложения возможность продолжать работу, даже если эта часть приложения окажется перегруженной, и использовать все потоки исполнителя `custom-name`.

Далее рассмотрим конфигурацию специального сервиса-исполнителя.

Рукопожатия и сбросы

Другой вариант гибкой связи — *рукопожатия* и *замедленная обратная реакция*. Суть его в том, что партнер по связи, находящийся под нагрузкой, уведомляет другую сторону, которая затем отступает и уменьшает нагрузку. Рукопожатие в данном случае означает, что у вызывающей стороны есть способ запросить сервис, может ли он обработать следующие запросы. Замедленная обратная реакция снижает нагрузку на систему, уведомляя клиентов о достижении предела или возвращая запросы.

Сочетание этих двух методов позволяет построить устойчивую и эффективную систему связи.

Информация о текущей нагрузке на приложение может быть предоставлена через HTTP-ресурсы или поля заголовка, после чего клиенты могут ее учитывать в дальнейшем.

Более короткий путь — просто отклонить запрос клиента, если ресурсы сервера полностью исчерпаны. Программистам рекомендуется обратить внимание на поведение концентратора, например, в сервисах-исполнителях и на то, как они обрабатывают ситуации с переполненными очередями. В исключительных случаях рекомендуется отбросить запрос клиента, чтобы он не вызывал дополнительных задержек.

В следующем примере показан сценарий с использованием библиотеки Porcupine. Бизнес-функция выполняется с применением выделенного сервиса-исполнителя, конфигурация которого предусматривает отмену отклоненных исполнений. Клиенты будут сразу получать ответ `503 Service Unavailable`, показывающий, что в настоящий момент сервис не может обслуживать запросы.

Ресурс JAX-RS здесь задействован, как в предыдущем примере. Конфигурация исполнителя `custom-name` предусматривает отмену отклоненных исполнений посредством специализированного конфигуратора. `ExecutorConfigurator` является частью библиотеки Porcupine. Далее приведена специальная конфигурация:

```
import com.airhacks.porcupine.configuration.control.ExecutorConfigurator;
import com.airhacks.porcupine.execution.control.ExecutorConfiguration;

@Specializes
public class CustomExecutorConfigurator extends ExecutorConfigurator {

    @Override
    public ExecutorConfiguration defaultConfigurator() {
        return super.defaultConfigurator();
    }

    @Override
    public ExecutorConfiguration forPipeline(String name) {
        if ("custom-name".equals(name)) {
            return new ExecutorConfiguration.Builder().
                    abortPolicy().
                    build();
        }
        return super.forPipeline(name);
    }
}
```

Исполнения, которые были отклонены из-за переполнения очереди, в дальнейшем приводят к исключению `RejectedExecutionException`. Оно преобразуется через функциональность JAX-RS следующим образом:

```
import java.util.concurrent.RejectedExecutionException;

@Provider
public class RejectedExecutionHandler
        implements ExceptionMapper<RejectedExecutionException> {

    @Override
    public Response toResponse(RejectedExecutionException exception) {
        return Response.status(Response.Status.SERVICE_UNAVAILABLE)
                .build();
    }
}
```

Запросы клиентов, превышающие возможности сервера, сразу вызывают сообщение об ошибке. Клиентский вызов может принять это во внимание и действовать соответствующим образом. Например, функция шаблона, играющая роль предохранителя, может препятствовать немедленному повторению клиентского вызова.

Замедленная обратная реакция оказывается полезной при разработке сценариев с несколькими сервисами, которые должны соответствовать *соглашениям об уровне обслуживания (service level agreements, SLA)*. Подробнее эта тема будет рассмотрена в главе 9.

Подробнее об устойчивости

Помимо обеспечения устойчивой связи, микросервисы предназначены для повышения качества и доступности сервисов. Приложения должны иметь возможность масштабирования и самовосстановления в случае сбоев.

Использование технологии управления контейнерами, такой как Kubernetes, поддерживает эту концепцию. Модули, возвращающие логические сервисы, могут быть расширены, чтобы выдерживать большую нагрузку. Сервисы распределяют нагрузку между контейнерами. Есть возможность автоматически увеличивать и уменьшать количество экземпляров в зависимости от текущей рабочей нагрузки в кластере.

Kubernetes стремится максимально увеличить время доступности сервиса. Он считывает значения датчиков жизнедеятельности и готовности, чтобы вовремя обнаруживать сбои и при необходимости запускать новые контейнеры. В случае ошибок во время развертывания Kubernetes сделает так, чтобы это не коснулось уже работающих сервисов, пока новые версии не смогут обслуживать трафик.

Эти методы управляются не приложением, а средой выполнения. В корпоративном приложении рекомендуется свести к минимуму нефункциональные сквозные задачи.

Резюме

Есть множество причин для использования распределенных систем. Несмотря на некоторые описанные здесь проблемы и затраты на коммуникацию, потерю производительности и организационные проблемы, распределенная система часто является необходимостью.

При проектировании системной среды следует учитывать системную карту контекстов, которая представляет отдельные задачи. Целесообразно разрабатывать API приложения в виде простых и надежных интерфейсов, в идеале реализованных на базе стандартных протоколов связи. Прежде чем вводить изменения, инженеры, а также бизнес-эксперты должны спросить себя, так ли уж необходимо принудительно прерывать работу клиентских функций. К тому же API следует делать гибкими во избежание ненужных остановок в работе, другими словами, быть консервативными в том, что вы делаете, и либеральными в том, что принимаете.

Инженеры, создающие распределенные приложения, должны найти компромисс между согласованностью и масштабируемостью. Большинство приложений, использующих синхронную связь с внешней системой, скорее всего, будут довольно хорошо масштабироваться. Следует избегать распределенных транзакций.

Для обеспечения асинхронной связи приложение может быть основано на архитектуре с регистрацией событий. Принцип CQRS сочетает в себе свойства событийно-ориентированной архитектуры и регистрации событий. CQRS, безусловно, предлагает интересные решения, но это имеет смысл только в том случае, если есть необходимость в создании распределенной системы.

Микросервисные архитектуры не должны иметь совместно используемых технологий или данных. Архитектуры без разделения ресурсов предоставляют свободу выбора технологии реализации и сохранения данных. Приложения Java EE с нулевыми зависимостями, поставляемые в контейнерах, вполне подходят для создания микросервисов. Концепция «одно приложение — один сервер приложений» соответствует идее архитектуры без разделения ресурсов.

В следующей главе будут рассмотрены вопросы производительности, мониторинга и журналирования.

9 Мониторинг, производительность и журналирование

Мы уже знаем, как создавать современные масштабируемые и отказоустойчивые микросервисы на Java EE. А сейчас продолжим тему обеспечения устойчивой связи, а также решения технических сквозных задач при разработке микросервисов.

Корпоративные приложения работают в серверных средах, удаленных от пользователей. Для того чтобы обеспечить понимание системы, необходимо сделать их видимыми. Существует несколько способов решения этой телеметрической задачи, которые включают в себя мониторинг, проверку работоспособности, отслеживание и журналирование. В этой главе рассматриваются причины, по которым выбирают тот или иной метод, и показатели, имеющие смысл для корпоративных приложений.

В этой главе будут рассмотрены следующие темы.

- Бизнес-показатели и технические показатели.
- Интеграция Prometheus.
- Как удовлетворить потребность в производительности.
- Диагностическая модель производительности Java.
- Методы мониторинга и организация выборочных исследований.
- Почему традиционное журналирование вредно.
- Мониторинг, журналирование и отслеживание в современном мире.
- Пригодность тестов производительности.

Бизнес-показатели

Видимость в бизнес-процессах имеет решающее значение для участников бизнеса, которым нужно видеть и интерпретировать происходящее внутри корпоративной системы. Связанные с бизнесом показатели позволяют оценивать эффектив-

ность процессов. Если процессы невидимы, корпоративное приложение подобно черному ящику.

Показатели, связанные с бизнесом, являются бесценным активом для бизнес-экспертов. Они предоставляют информацию о предметной области и выполнении бизнес-сценариев. Какие показатели интересны, зависит от предметной области.

Сколько автомобилей создается в час? Сколько товаров куплено и на какую сумму? Каков курс валют? Сколько клиентов пришло вследствие маркетинговой кампании по электронной почте? Все это примеры ключевых показателей эффективности для своих областей. Показатели для конкретной области определяют бизнес-эксперты.

Корпоративное приложение должно предоставлять информацию, которая поступает из разных точек бизнес-процессов. Характер этой информации зависит от конкретной предметной области. Во многих случаях бизнес-показатели обусловлены событиями предметной области, происходящими во время выполнения бизнес-процессов.

Возьмем, к примеру, количество автомобилей, выпускаемых за час. Бизнес-сценарий создания автомобиля генерирует соответствующее событие предметной области `CarCreated`, которое собирают для будущей статистики. А, например, расчет курса валют потребует гораздо больше информации.

Бизнес-эксперты должны определить семантику и происхождение ключевых показателей эффективности. Определение и сбор этих показателей становится частью бизнес-сценария. За выдачу данной информации также отвечает приложение.

Важно различать показатели, собираемые для бизнеса и по соображениям технологии. Бизнес-показатели дают представление о ценности приложения, но на них напрямую влияют технические показатели. Примером технического показателя является время отклика сервиса, которое, в свою очередь, зависит от других технических показателей. Этот вопрос будет подробно рассмотрен в разделе «Технические показатели».

Поэтому бизнес-эксперты должны интересоваться не только бизнес-аспектами мониторинга, но и техническими результатами работы приложения.

Сбор бизнес-показателей

Показатели, относящиеся к бизнесу, позволяют специалистам в данной области оценивать эффективность корпоративной системы. Они обеспечивают получение полезных сведений о конкретных сторонах данного бизнеса. Приложение отвечает за сбор показателей, связанных с бизнесом, в рамках его бизнес-сценариев.

Например, приложение по производству автомобилей реализует бизнес-логику, которая может выдавать такие показатели, как количество автомобилей, создаваемых за час.

С точки зрения бизнеса соответствующие показатели обычно обусловлены событиями предметной области. Целесообразно определять и генерировать события предметной области, такие как `CarCreated`, в рамках бизнес-сценария после успешного изготовления автомобиля. Эти события собирают и используют для получения дополнительной информации в виде конкретных бизнес-показателей.

Событие `CarCreated` генерируется в контуре как CDI-событие и может отслеживаться специальным сборщиком статистики. В следующем фрагменте кода показано событие предметной области, генерируемое в процессе выполнения бизнес-сценария:

```
@Stateless
public class CarManufacturer {

    @Inject
    CarFactory carFactory;

    @Inject
    Event<CarCreated> carCreated;

    @PersistenceContext
    EntityManager entityManager;

    public Car manufactureCar(Specification spec) {
        Car car = carFactory.createCar(spec);
        entityManager.merge(car);
        carCreated.fire(new CarCreated(spec));
        return car;
    }
}
```

Контур запускает CDI-событие, которое сообщает об успешном создании автомобиля. Соответствующая обработка отделена от бизнес-процесса, поэтому здесь больше нет никакой логики. Событие будет отслеживаться другим компонентом, включенным в состав приложения. Синхронные CDI-события могут создаваться для обработки на определенных этапах транзакций. Таким образом, следующий наблюдатель транзакций гарантирует, что будут засчитаны только успешные транзакции базы данных:

```
import javax.enterprise.event.TransactionPhase;

@ApplicationScoped
public class ManufacturingStatistics {

    public void carCreated(@Observes(during =
            TransactionPhase.AFTER_SUCCESS) Specification spec) {
        // сбор статистики о производстве автомобилей
        // в соответствии с заданной спецификацией,
        // то есть увеличение счетчиков
    }
}
```

Информацию о событиях собирают и обрабатывают, чтобы получить бизнес-показатели. В зависимости от ситуации могут потребоваться дополнительные данные, относящиеся к сфере бизнеса.

Представление необходимой информации в виде событий предметной области соответствует определению бизнеса и отделяет бизнес-сценарий от вычисления статистики.

В зависимости от ситуации и требований информацию можно собирать не только как события предметной области, но и с помощью сквозных компонентов, таких как перехватчики. В простейшем случае показатели измеряются и собираются в базовых элементах. Разработчики приложений должны учитывать области видимости компонентов, чтобы собранные данные не отбрасывались из-за неправильно выбранных областей видимости.

Выдача показателей

Показатели обычно сохраняются не в приложении, а в другой системе, которая является частью среды выполнения, например во внешней системе мониторинга. Это упрощает реализацию системы показателей, а корпоративное приложение хранит информацию в памяти и выдает заданные показатели. Внешние системы мониторинга получают и обрабатывают их.

Есть несколько методов, которые можно использовать для выдачи и сбора показателей. Например, они могут иметь формат специализированных строк JSON и передаваться через конечные точки HTTP.

В настоящее время очень популярна система мониторинга *Prometheus*, входящая в состав Cloud Native Computing Foundation. Prometheus — это технология мониторинга и оповещения, которая получает, эффективно хранит и запрашивает временные данные. Она собирает данные, выдаваемые другими сервисами через HTTP в определенном формате. Prometheus имеет широкие возможности получения и хранения данных.

Для построения графиков и информационных панелей с отображением бизнес-информации можно использовать другие решения на базе Prometheus. Одна из технологий, хорошо совместимая с Prometheus и предоставляющая множество возможностей для создания наглядных графиков, называется *Grafana*. Сама она не хранит временные данные, но применяется для запросов и отображения временных данных из таких источников, как Prometheus.

На рис. 9.1 показан пример информационной панели Grafana.

Информационные панели объединяют релевантную информацию и обеспечивают ее визуализацию для бизнес-экспертов. В зависимости от требований и причин связанная информация компонуется в виде графиков, которые позволяют получить общую картину и определенные сведения. Информационные панели дают возможность запрашивать и настраивать представления временных данных в зависимости от целевой группы.

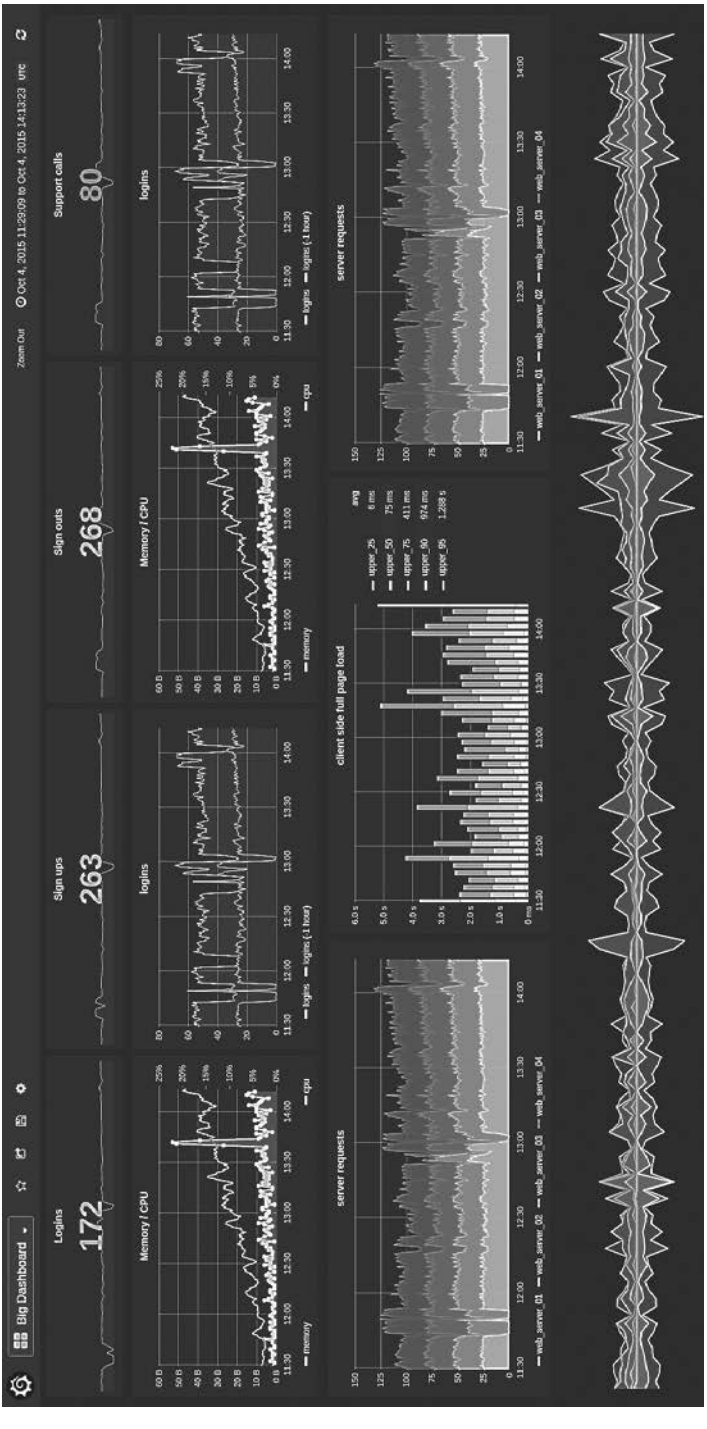
Big Dashboard
Zoom Out
Oct 4, 2015 11:29:09 to Oct 4, 2015 14:13:23 utc
Logins
172
Sign ups
263
Sign outs
268
Support calls
80
Memory / CPU
logins
server requests
client side full page load

Рис. 9.1

Введение в Prometheus

В следующих примерах показано, как интегрировать Prometheus в Java EE. Это одно из возможных решений для мониторинга, и мы хотим дать читателям представление о том, как плавно интегрировать показатели, связанные с бизнесом.

Приложение будет выдавать собранные показатели в выходном формате Prometheus. Экземпляры Prometheus получают и хранят эту информацию, как показано на рис. 9.2.

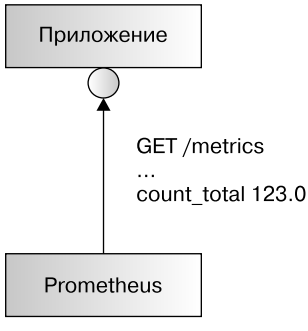

Рис. 9.2

Программисты могут реализовать дополнительные функции для сбора и выдачи информации или использования клиентского API Prometheus, который поставляется с несколькими готовыми типами показателей.

В Prometheus есть несколько типов показателей, в том числе следующие.

- *Счетчик* — один из самых распространенных показателей, представляющий собой увеличивающееся числовое значение. Он подсчитывает произошедшие события.
- *Шкала* — числовое значение, которое может увеличиваться и уменьшаться. Его можно использовать для измерения таких значений, как курс валют, температура или товарооборот.
- *Гистограммы* и *сводки* — более сложные типы показателей, применяемые для выборочных наблюдений на определенных временных отрезках. Обычно они отслеживают распределение показателей, например, сколько времени занимает выпуск автомобиля, насколько эти значения различаются и как они распределяются.

У каждого показателя Prometheus есть имя и метки, которые представляют собой пары «ключ — значение». Временной ряд идентифицируется по имени показателя и набору меток. Метку можно рассматривать как множество параметров, сегментирующее информацию.

Примером представления показателя «счетчик» с использованием меток является следующий: `cars_manufactured_total {color = "RED", engine = "DIESEL"}`. Счетчик `cars_manufactured_total` показывает общее количество изготовленных

автомобилей с заданными цветом и типом двигателя. Впоследствии собранные показатели могут быть выданы по запросу с указанием информации о метках.

Реализация в Java EE

В следующем примере показана реализация сбора статистики с отслеживанием описанного ранее события предметной области и сохранения информации в показателе Prometheus «счетчик»:

```
import io.prometheus.client.Counter;

@ApplicationScoped
public class ManufacturingStatistics {

    private Counter createdCars;

    @PostConstruct
    private void initMetrics() {
        createdCars = Counter.build("cars_manufactured_total",
                "Всего произведено автомобилей ")
                .labelNames("color", "engine")
                .register();
    }

    public void carCreated(@Observes(during =
            TransactionPhase.AFTER_SUCCESS) Specification spec) {
        createdCars.labels(spec.getColor().name(),
                spec.getEngine().name()).inc();
    }
}
```

Счетчик создается и регистрируется в клиентском API Prometheus. Измеряемые значения зависят от значений свойств автомобиля `color` и `engine`, которые учитываются при получении значений.

Для того чтобы выдавать эту информацию, можно подключить библиотеку сервлетов Prometheus. Она выводит все зарегистрированные показатели в правильном формате. Конфигурация сервлета мониторинга определяется через `web.xml`. Также можно подключить ресурс JAX-RS, чтобы выдавать данные посредством обращения к `CollectorRegistry.defaultRegistry`.

Выданные приложением выходные данные будут выглядеть примерно так:

```
...
cars_manufactured_total{color="RED", engine="DIESEL"} 4.0
cars_manufactured_total{color="BLACK", engine="DIESEL"} 1.0
```

Компоненты Java EE, такие как CDI-события, упрощают работу программистов в том, что плавно интегрируют показатели событий предметной области. В предыдущем примере класс `ManufacturingStatistics` был единственным элементом, который зависел от API Prometheus.

Очень желательно подключать клиентский API Prometheus как отдельный уровень образа контейнера, а не включать его в артефакт приложения.

Решение для мониторинга получает и затем обрабатывает предоставленную информацию, чтобы собрать необходимые бизнес-показатели. Получение значений счетчика изготовленных автомобилей в зависимости от времени даст количество автомобилей, создаваемых в час. Этот показатель можно запрашивать, чтобы узнать общее количество автомобилей, а также количество автомобилей определенного цвета и с нужным типом двигателя. Запросы, определяющие бизнес-показатели, также могут адаптироваться и уточняться в соответствии с текущими требованиями. В идеале приложение выдает необходимые атомарные бизнес-показатели.

Интегрирование в среду выполнения

Приложение выдает показатели, относящиеся к бизнесу, через HTTP. Экземпляр Prometheus получает и сохраняет эти данные и делает их доступными посредством запросов, графиков и внешних решений, таких как Grafana.

Во фреймворке управления контейнером экземпляр Prometheus запускается внутри кластера. Это избавляет от необходимости настраивать доступные извне конечные точки мониторинга. Для того чтобы обнаружить экземпляры приложения, Prometheus интегрируется с Kubernetes. Prometheus нуждается в доступе к каждому модулю приложения в отдельности, так как каждый экземпляр приложения выпускает свои показатели мониторинга. Prometheus накапливает информацию обо всех экземплярах.

Конфигурация Prometheus хранится либо в карте конфигурации, либо как часть базового образа. Экземпляр настроен так, чтобы каждые *n* секунд обращаться к приложениям и экспортерам для получения временных рядов. Подробнее о конфигурации Prometheus читайте в актуальной версии документации.

Это только одно из возможных решений для интеграции бизнес-мониторинга в облачное приложение.

Показатели, связанные с бизнесом, целесообразно представлять в виде событий предметной области, которые возникают в процессе выполнения бизнес-сценария. Интеграция выбранных решений для мониторинга должна происходить прозрачно для логики предметной области. К тому же не должно возникать значительной зависимости от сторонних продуктов.

Требования к производительности в распределенных системах

Время реакции является важным нетехническим требованием к корпоративному приложению. Система будет представлять ценность для бизнеса только в том случае, если запросы клиентов обслуживаются за разумное время.

Для того чтобы удовлетворить требования к производительности, в распределенных системах необходимо учитывать все задействованные приложения.

Корпоративное приложение часто должно выполнять *соглашение об уровне обслуживания (service level agreement, SLA)*. В SLA обычно определены пороговые значения доступности или времени отклика.

Соглашения об уровне обслуживания

Для того чтобы рассчитать и выполнить все SLA, важно определить, какие процессы и приложения участвуют в бизнес-сценариях, особенно в синхронной связи. Скорость работы приложений, которые синхронно вызывают внешние системы, напрямую зависит от скорости этих вызовов. Как уже упоминалось, следует избегать распределенных транзакций.

По своей природе SLA могут быть выполнены только в том случае, если все приложения работают и хорошо сочетаются между собой. Каждое приложение влияет на SLA зависимых систем. Это касается не только самого медленного приложения в системе, но и всех задействованных сервисов.

Например, удовлетворить требование безотказной работы в 99,995 % случаев невозможно, если оно включает в себя синхронные вызовы двух приложений, каждое из которых гарантирует 99,995 %. В результате SLA составит 99,99 % — значения для каждой участвующей системы увеличиваются.

То же самое верно для гарантированного времени отклика. Каждая задействованная система замедляет общее время отклика, в результате чего оно будет равно сумме времени для всех SLA.

Вычисление SLA в распределенной системе

Рассмотрим пример того, как выполнить SLA в распределенных системах, если корпоративное приложение является частью сценария с высокой нагрузкой и критически важно обеспечить гарантированное время отклика. Приложение синхронно связывается с одной или несколькими серверными системами, предоставляющими необходимую информацию. Вся система в целом согласно SLA должна обеспечивать время отклика 200 мс.

В этом сценарии серверные приложения соответствуют требованиям к времени отклика SLA за счет замедленной обратной реакции и превентивного отказа от запросов, которые не выполняют условия SLA. При этом у исходного приложения есть возможность использовать другой серверный сервис, который успеет отреагировать вовремя.

Для того чтобы правильно настроить регулирование количества запросов, инженерам нужно знать среднее время отклика серверной системы. В данном случае это 20 мс. Соответствующая бизнес-функция определяет выделенный пул потоков с помощью специального сервиса управляемых исполнителей. Конфигурация пула потоков настраивается индивидуально.

Настройка конфигурации происходит следующим образом: инженеры выбирают максимальный размер пула потоков и максимальный размер очереди так, чтобы время SLA в n раз превышало среднее время отклика. В данном случае

n равно 10 — это максимальное количество запросов, которые система будет обрабатывать одновременно. Оно равно сумме максимального размера пула и максимального размера очереди. Если это количество будет превышено, то любой запрос немедленно будет отклонен сервисом с выдачей сообщения о временной недоступности. В основе этой логики лежит такой расчет: если текущее количество обрабатываемых запросов превышает *n*, то новый запрос превысит расчетное время SLA в 200 мс.

«Немедленное отклонение запросов» звучит жестко, но при этом клиентскому приложению предоставляется возможность обратиться к другому серверному приложению, не тратя все время, предусмотренное SLA, на один вызов. Это пример сценария высокой производительности с несколькими серверными приложениями, где выполнение условий SLA имеет высокий приоритет.

Реализация этого сценария аналогична примеру с замедленной обратной реакцией, описанному в предыдущей главе. Если первый вызов завершился неудачей из-за недоступного сервиса, клиент может обратиться к другому серверному приложению. Это неявно обеспечивает гибкость клиента, поскольку он использует несколько резервных серверных приложений. Для серверной части неявно применяется шаблон переборки. Если одна из функций оказывается недоступной, это не влияет на остальную часть приложения.

Решение проблем производительности

Технические показатели, такие как время отклика, пропускная способность, частота ошибок или время безотказной работы, говорят о скорости реакции системы. Пока скорость реакции приложения находится в допустимых пределах, пожалуй, никакие другие показатели не нужны. Недостаточная производительность означает, что SLA системы не выполняется, то есть время ответа слишком велико или клиентские запросы заканчиваются неудачей. Возникает вопрос: что нужно изменить, чтобы улучшить ситуацию?

Теория ограничений

Если желаемая нагрузка на систему увеличивается, пропускная способность в идеале также возрастает. Теория ограничений основана на предположении, что существует хотя бы одно ограничение, которое будет тормозить пропускную способность системы. Поэтому ограничения или узкие места приводят к падению производительности.

Подобно цепи, прочность которой определяется самым слабым звеном, ограничивающий ресурс сдерживает общую производительность системы или ее определенные функциональные возможности. Таким образом, приложение не может справиться с большей нагрузкой, пока другие ресурсы не используются в полной мере. Только увеличивая поток через ограничивающий ресурс, то есть устраняя узкое место, можно увеличить пропускную способность. Если оптимизировать

всю систему, *кроме узкого места*, вместо того чтобы его удалить, то время ее реакции не только не уменьшится, но даже может увеличиться.

Поэтому критически важно определить, где находится узкое место. Пока не будет преодолено это ограничение, общая производительность не улучшится.

Например, выделение большей мощности процессора для приложения с его значительной загрузкой едва ли поможет обеспечить более высокую производительность. Возможно, приложение плохо работает из-за других, глубинных причин, а не из-за недостаточной мощности процессора.

Здесь важно отметить, что ограничивающие условия, скорее всего, являются внешними по отношению к приложению. В едином монолитном приложении они включают в себя аппаратные средства и операционную систему со всеми запущенными в ней процессами. Если другое программное обеспечение, работающее на том же оборудовании, интенсивно использует сетевой адаптер, то сетевые операции ввода/вывода и общая производительность приложения будут от этого зависеть, даже если основное ограничивающее условие не входит в компетенцию приложения.

Поэтому для проверки проблем производительности необходимо учитывать больше, чем само приложение. На производительность приложения может влиять весь набор процессов, работающих на том же оборудовании, в зависимости от того, как другие процессы используют системные ресурсы.

В распределенной среде аналитика производительности включает в себя все взаимозависимые приложения, которые взаимодействуют с данным приложением, и связь между ними. Для того чтобы определить ограничивающий ресурс, необходимо рассмотреть ситуацию в системе в целом.

Поскольку приложения взаимосвязаны, повышение скорости реакции одной системы повлияет на другие и может снизить скорость реакции всей системы. Опять же неправильный выбор показателя, нуждающегося в улучшении, например оптимизация всего, за исключением узкого места, не повысит, а скорее даже снизит общую производительность. Приложение, подключающееся к внешней системе, которая представляет собой узкое место, оказывает определенное давление на внешнюю систему. Если оптимизировать производительность приложения, а не внешней системы, то нагрузка на внешнюю систему и давление на нее увеличатся, что в итоге приведет к увеличению общего времени реакции.

В распределенных системах ситуация со всеми взаимозависимыми приложениями значительно усложняет решение проблем производительности.

Определение падения производительности с помощью jPDM

Java Performance Diagnostic Model Java (jPDM) — это диагностическая модель производительности, которая абстрагирует сложность систем. Это помогает интерпретировать счетчики производительности системы и, таким образом, понять основную причину ее падения.

Проблема определения причины падения производительности заключается в том, что конкретный сценарий складывается в результате многочисленных воздействий, многие из которых являются внешними по отношению к приложению. jPDM и полученные с ее помощью методики помогают решить эту задачу.

С точки зрения скорости реакции существует бесконечно много моментов, которые могут вызывать проблемы, но все же количество проблем является конечным. Таким образом, причины падения производительности могут быть отнесены к конечному количеству категорий. Есть несколько типичных проблем, вызванных бесчисленным количеством различных сценариев и глубинных причин. Для того чтобы определить эти категории, будем использовать диагностическую модель.

jPDM определяет важные подсистемы исследуемой системы, их роли, функции и атрибуты. Эти подсистемы взаимодействуют между собой. Модель помогает выбрать инструменты для измерения уровней активности и взаимодействия между подсистемами. Методы и процессы, помогающие изучать и анализировать системы и ситуации в смысле производительности, выходят за рамки данной модели.

Подсистемы

Подсистемами в прикладной среде Java являются операционная система, включая аппаратное обеспечение, виртуальная машина Java, приложение и акторы. Каждая подсистема задействует подчиненные ей подсистемы для выполнения своих задач.

На рис. 9.3 показано, как подсистемы jPDM взаимодействуют между собой.

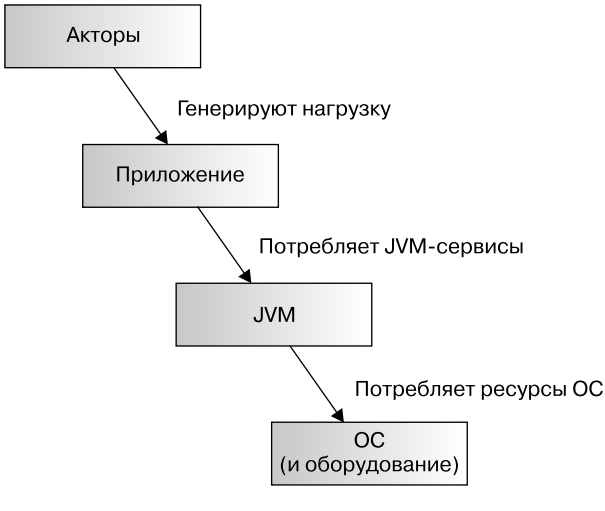

Рис. 9.3

Акторы

Акторы — это пользователи системы в самом широком смысле: конечные пользователи, пакетные процессы и внешние системы — в зависимости от бизнес-сценария.

Используя систему, акторы генерируют рабочую нагрузку. Свойствами акторов являются коэффициент нагрузки, то есть количество задействованных пользователей, а также скорость, то есть то, как быстро обрабатываются пользовательские запросы. Эти свойства, как и другие свойства подсистемы, влияют на общую ситуацию.

Сами акторы не создают проблемы производительности, они просто используют приложение. Другими словами, если производительность системы неудовлетворительна, то не стоит искать ограничивающее условие на уровне акторов — они и создаваемая ими загрузка являются частью условий, в которых должна работать система.

Приложение

Корпоративное приложение содержит алгоритмы бизнес-логики. Частью бизнес-сценариев являются выделение памяти, планирование потоков и применение внешних ресурсов.

Для этого приложение задействует функциональные возможности фреймворка и языка Java. В конечном счете оно явно или неявно использует код и конфигурацию JVM. Таким образом, приложение создает определенную нагрузку на JVM.

JVM

Виртуальная машина Java (Java Virtual Machine, JVM) интерпретирует и выполняет байт-код приложения. Она берет на себя управление памятью — выделение памяти и сбор мусора. Существуют обширные методы оптимизации для повышения производительности программ, такие как *компиляция Just-In-Time (JIT)* в Java HotSpot Performance Engine.

Для распределения памяти, запуска потоков, сетевого или дискового ввода/вывода JVM использует ресурсы операционной системы.

Операционная система и оборудование

Аппаратные компоненты компьютера, такие как процессор, память, диск и сетевой адаптер, определяют ресурсы системы. Они имеют определенные свойства, такие как емкость или скорость.

Поскольку аппаратные компоненты представляют собой ресурсы, недоступные для совместного применения, операционная система распределяет аппаратные средства между процессами. Операционная система обеспечивает доступ к общесистемным ресурсам и распределяет потоки между процессорами.

По этой причине модель рассматривает общую систему, включая аппаратное обеспечение. Корпоративное приложение, скорее всего, является не единственным потребителем аппаратных ресурсов. Другие процессы также используют аппарат-

ные компоненты и, таким образом, влияют на производительность приложения. Попытка нескольких процессов одновременно получить доступ к сети приведет к тому, что скорость отклика будет ниже, чем для каждого из них в отдельности.

Экземпляры jPDM в условиях эксплуатации

Экземпляры модели jPDM описывают конкретные ситуации, возникающие в процессе эксплуатации. Они содержат все их свойства, характеристики и специфические узкие места.

Любое изменение одной из подсистем приведет к другой ситуации, с другими свойствами и характеристиками и, следовательно, к другому экземпляру модели. Например, изменение нагрузки на систему может вызвать появление совершенно другого узкого места.

Это одна из причин того, почему тесты производительности в средах, отличных от среды эксплуатации, могут указывать на совсем другие узкие места. В иной среде как минимум иная ситуация с ОС и аппаратным обеспечением, и дело не столько в используемом оборудовании и конфигурации, сколько в состоянии ОС в целом. Таким образом, имитируемые сценарии, такие как тесты производительности, не позволяют делать выводы об узких местах и оптимизации производительности, так как представляют собой другой экземпляр jPDM.

Поскольку мы в конечном счете применяем модель для анализа проблем с производительностью, описанные далее методы имеют смысл только тогда, когда возникают такие проблемы. Если их нет, то есть заданные SLA выполняются, то нечего исследовать и не на что воздействовать.

Анализ экземпляров jPDM

jPDM используется для исследования падения производительности. Методики, процессы и инструменты, которые определяются моделью, помогают выявить ограничивающие условия.

Каждая подсистема со своим набором атрибутов и ресурсов играет собственную роль в системе. Мы применяем специальные инструменты, чтобы выявить конкретные показатели производительности и проследить взаимодействие между подсистемами.

В соответствии с теорией ограничений мы хотим исследовать ограничивающие условия среды эксплуатации как экземпляра jPDM. Инструментарий помогает в этом. Важно, чтобы исследование рассматривало систему в целом. Аппаратные средства используются всеми процессами операционной системы. Следовательно, доминирование может быть вызвано одним процессом или суммой всех процессов, выполняемых на данном оборудовании.

Прежде всего мы исследуем доминирующего потребителя ресурсов процессора и то, как используется последний. Шаблон потребления ресурсов CPU приведет нас к подсистеме, которая является узким местом.

Для того чтобы исследовать доминирующего потребителя, воспользуемся деревом решений. Оно показывает, где расходуется больше всего процессорного

времени — в пространстве ядра, пространстве пользователя или режиме ожидания. Дерево решений показано на рис. 9.4.

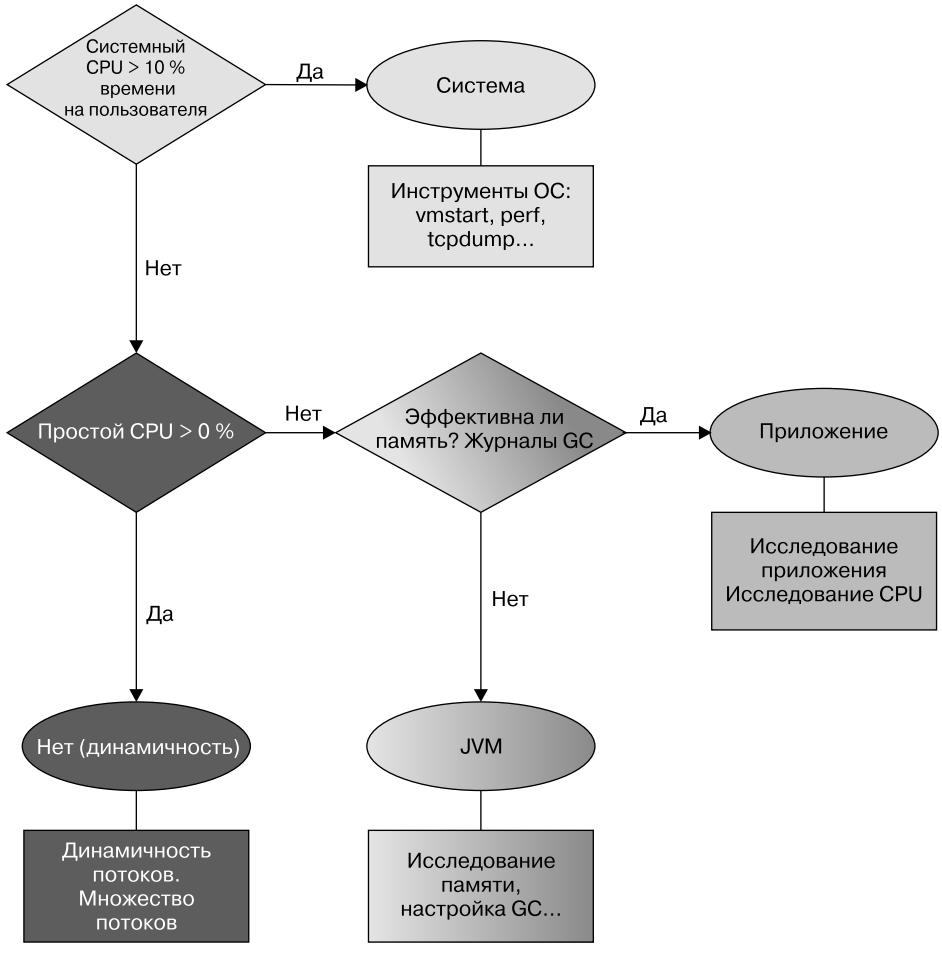

Рис. 9.4

В овальных рамках на диаграмме представлены доминирующие потребители ресурсов процессора, цвета соответствуют подсистемам jPDM. Проходя по дереву решений, мы придем к подсистеме, в которой есть узкое место. Идентификация подсистемы сужает круг систем, которые могли вызвать падение производительности, до определенной категории. Затем мы используем дополнительные инструменты для анализа реальной ситуации в экземпляре jPDM.

Поскольку у падения производительности может быть бесконечное множество причин, нужно следовать описанному процессу, чтобы сузить круг возможных вариантов. Если бы мы не шли по дереву вариантов, а просто слепо действовали *методом тыка* или пытались угадать, то не только зря тратили бы время и силы,

но и могли бы ошибочно принять определенные признаки за фактически доминирующие ограничения.

Доминирующие потребители ресурсов CPU — это именно то, на что расходуется процессорное время. Это важная информация для исследования ситуации. Недостаточно просто рассмотреть общий объем использования CPU. Эта информация сама по себе не свидетельствует о наличии узкого места или запасе ресурса процессора и не указывает на доминирующего потребителя. Задействование процессора на 60 % не говорит о том, является ли процессор ограничивающим ресурсом и улучшится ли общее время отклика, если увеличить количество CPU. Необходимо проанализировать время CPU более подробно.

Прежде всего нужно изучить соотношение между временем CPU, затраченным на обслуживание пользователей и системы. Это покажет, расходуется ли на задачи ядра больше процессорного времени, чем ожидалось, и, следовательно, является ли операционная система доминирующим потребителем ресурсов процессора.

Доминирующий потребитель — ОС

Операционная система доминирует при потреблении ресурсов процессора, когда требуется, чтобы он работал более интенсивно, чем обычно. Это означает, что слишком много процессорного времени расходуется на управление ресурсами и устройствами: сетевыми и дисковыми операциями ввода/вывода, блокировками, управлением памятью и контекстными переключателями.

Если процентное соотношение системного и пользовательского времени процессора превышает определенное значение, это значит, что операционная система является доминирующим потребителем. jPDM считает 10 % пороговым значением, основываясь на опыте анализа множества производственных ситуаций. Это означает, что если системное время процессора составляет более 10 % от пользовательского времени, то подсистема ОС является узким местом.

В этом случае нужно исследовать проблему дальше, используя инструменты операционной системы, такие как `vmstat`, `perf`, `netstat` и др. Например, если корпоративное приложение, извлекает записи из базы данных, генерируя огромное количество индивидуально выполняемых запросов, то управление всеми этими подключениями к базе данных создает значительную нагрузку на операционную систему. Расходы на установление каждого сетевого соединения в итоге будут доминировать в системе и станут ограничивающим системным ресурсом. Таким образом, исследование ситуации покажет, что значительная часть процессорного времени приходится на ядро, где устанавливаются сетевые подключения.

Отсутствие доминирующего потребителя

Если исследование расхода процессорного времени не определило ОС в качестве доминирующего потребителя, то в соответствии с деревом решений следует проанализировать, не простаивает ли процессор. Если это так, значит, есть процессорное время, которое невозможно использовать.

Поскольку мы анализируем ситуацию, когда SLA не выполняется, то есть общая система недостаточно эффективна для данной нагрузки, то в режиме полного

насыщения процессор должен задействоваться полностью. Следовательно, простой процессора указывает на проблему динамичности.

В этом случае нужно исследовать, почему операционная система не составила расписание потоков. Причин может быть несколько: пустые соединения или пулы потоков, взаимная блокировка или медленные отклики внешних систем. На причину ограничения будет указывать состояние потоков. Для исследования этой ситуации можно снова воспользоваться инструментарием операционной системы.

Примером проблем данной категории является замедленная реакция внешней системы, к которой выполняется синхронный доступ. Это приведет к появлению потоков, которые не могут работать, так как ожидают ответа от сетевого ввода/вывода. В отличие от доминирующего потребления ОС, здесь поток не работает активно, а ждет, когда он будет запланирован.

Доминирующий потребитель — JVM

До сих пор мы рассматривали ситуации, когда доминирующие потребители не принадлежали подсистемам приложения или JVM. Если на процессы ядра и простои тратится не очень много процессорного времени, то мы начинаем исследование JVM.

Поскольку виртуальная машина Java отвечает за управление памятью, то ее производительность может указывать на проблемы с памятью. Исследование сценариев выполняется в основном с помощью журналов *сборки мусора (Garbage Collection, GC)* и инструментов *JMX*.

Утечки памяти увеличивают использование памяти и вызывают дополнительный запуск сборщиков мусора, которые создают нагрузку на процессор. Неэффективное использование памяти также приводит к дополнительному сбору мусора. Запуски GC в итоге приводят к тому, что JVM становится доминирующим потребителем ресурсов процессора.

Это еще один пример того, почему важно проводить исследования в соответствии с деревом решений jPDM. Проблема производительности выражается в высокой нагрузке на CPU, но в действительности узким местом в данном случае является утечка памяти.

Сейчас основная причина проблем с производительностью — нехватка памяти, в основном из-за ведения журналов приложений, что создает большое количество строковых объектов.

Доминирующий потребитель — приложение

Если анализ JVM не указывает на проблему с памятью, это значит, что доминирующим потребителем ресурсов процессора является приложение. Другими словами, за узкое место отвечает сам код приложения. В основном это касается приложений, которые реализуют сложные алгоритмы, чрезмерно нагружающие процессор.

Дальнейшее исследование приложений позволит сделать выводы о том, какая часть приложения создает проблему и как ее можно решить. Это означает, что

в приложении либо содержится неоптимальный код, либо оно достигло предела при имеющихся ресурсах и его нужно масштабировать по горизонтали или по вертикали.

Заключение

Способ решения проблем с производительностью заключается в том, чтобы сначала попытаться определить причины ее падения, исследуя ситуацию в соответствии с конкретным процессом. После идентификации ограниченного ресурса предпринимаются дальнейшие шаги по разрешению ситуации. После предположительного решения проблемы необходимо повторить измерение в среде эксплуатации. Важно не изменять поведение или конфигурацию, не убедившись, что изменения действительно приводят к ожидаемым результатам.

Метод jPDM исследует падение производительности без учета кода приложения, применяя единый процесс решения.

Какие инструменты и показатели необходимы для использования этого метода?

В зависимости от эксплуатируемой системы полезны инструменты, поставляемые в комплекте с операционной системой, а также инструменты Java Runtime. Поскольку при исследовании всех характеристик рассматривается вся среда выполнения на уровне операционной системы, а не только приложение, инструменты операционной системы и показатели более низкого уровня полезнее, чем инструменты исследования конкретных приложений.

Однако технические показатели приложения, такие как время отклика или пропускная способность, — это первое, на что следует обратить внимание, так как они говорят о качестве обслуживания приложения. Если эти показатели указывают на проблему производительности, то имеет смысл исследовать низкоуровневые показатели и инструменты.

В следующем разделе будет показано, как собирать технические показатели приложения.

Технические показатели

Технические параметры показывают нагрузку и скорость реакции системы. Основными из них являются время отклика и пропускная способность, обычно измеряемые в виде количества запросов или транзакций в секунду соответственно. Эти показатели предоставляют информацию о том, как работает вся система в данный момент.

Они влияют на другие связанные с бизнесом показатели. В то же время, как мы видели в предыдущем разделе, эти показатели являются всего лишь индикаторами и сами подвержены влиянию множества других технических компонентов, а именно всех свойств подсистем jPDM.

Поэтому производительность приложения определяется множеством технических воздействий. Возникает вопрос: какие технические показатели, помимо времени отклика, пропускной способности, частоты ошибок и времени безотказной работы, стоит собирать?

Типы технических показателей

Технические показатели, такие как время отклика или пропускная способность, относятся прежде всего к качеству обслуживания приложения. Они являются индикаторами, которые описывают скорость реакции приложения и могут указывать на потенциальные проблемы с производительностью. Эту информацию можно использовать для формирования статистики о тенденциях и пиковой нагрузке приложения.

Зная это, можно своевременно спрогнозировать потенциальные сбои и проблемы с производительностью. Это технический эквивалент бизнес-показателей, без которых система превращается в подобие черного ящика. Сами по себе данные показатели не позволяют сделать обоснованные выводы о первопричинах или ограничивающих ресурсах в случае проблем с производительностью.

Техническая информация низкого уровня включает в себя данные о потреблении ресурсов, потоках, пулах, транзакциях и сессиях. Еще раз отмечу: сама по себе она не указывает инженерам потенциальные узкие места.

Как было показано ранее, необходимо проверить общую ситуацию для всего программного обеспечения, которое работает на данном оборудовании. Наилучшим источником данных является информация об операционной системе. Для того чтобы решить проблемы с производительностью, нужно анализировать операционную систему и прикладные инструменты.

Это не означает, что техническая информация, полученная от приложения или среды выполнения JVM, бесполезна. Показатели конкретного приложения могут помочь решить проблемы с производительностью. Важно помнить, что сами по себе эти показатели могут заставить сделать неверные предположения о том, какие ресурсы являются ограничивающими при настройке производительности системы.

Высокочастотный мониторинг и выборочные исследования

Зачастую мониторинг направлен на сбор технических показателей с высокой частотой — много раз в секунду. Проблема заключается в том, что он сильно влияет на производительность системы. Показатели собирают часто даже в том случае, если падения производительности не наблюдается.

Как уже отмечалось, показатели на уровне приложения, такие как потребление ресурсов, сами по себе не помогают определить возможные ограничения производительности. Аналогично сбор информации снижает скорость реакции системы.

Рекомендуется собирать показатели реже, например несколько раз в минуту. Как показывает теория статистических совокупностей, эти несколько выборок довольно хорошо представляют всю совокупность данных.

Выборочные исследования информации должны как можно меньше влиять на производительность приложения. Все последующие исследования, будь то выборочное исследование показателей или вычисления, должны происходить за пределами системы, другими словами, передаваться внешней системе, которая не влияет на работающее приложение. Таким образом, выборочные исследования информации должны быть отделены от процессов ее хранения, запросов и отображения.

Сбор технических показателей

Приложение — это хорошее место для сбора технических показателей, в идеале на границах системы. Также можно собрать их на потенциальном прокси-сервере.

Сервер приложения сам генерирует технически релевантные показатели, такие как информация о потреблении ресурсов, потоках, пулах, транзакциях и сессиях. Некоторые решения также предоставляют Java-агентов, которые собирают и выдают технически релевантную информацию.

Традиционно на серверы приложений возлагается обязанность предоставлять технические показатели через JMX. Эта функциональность является частью API управления, но прежде редко использовалась в проектах. Одной из причин этого является то, что ее модель и API довольно громоздки.

Однако полезно напомнить, что серверы приложений Java EE должны собирать и предоставлять данные о своих ресурсах. Контейнер передает эту информацию через JMX. Есть несколько способов получить данную информацию.

Существуют так называемые экспортеры — приложения, которые работают либо автономно, либо как агенты Java. Они получают доступ к информации JMX и передают ее через HTTP. В качестве примера можно привести экспортер Prometheus JMX, который экспортирует информацию в аналогичном формате, как показано ранее. Преимущество этого подхода заключается в том, что он не добавляет зависимости в приложение.

Установка и настройка агентов Java выполняется на сервере приложений, на уровне образа базового контейнера. Это еще раз подчеркивает необходимость придерживаться принципа, согласно которому контейнеры не должны связывать артефакт приложения с деталями реализации.

Граничные показатели

Технические показатели, зависящие от приложения, такие как время отклика, пропускная способность, время безотказной работы и частота ошибок, могут быть собраны на границах системы через перехватчики или фильтры в зависимости от ситуации. Мониторинг через HTTP может осуществляться через сервлет-фильтр для любой технологии, основанной на сервлетах, такой как JAX-RS.

В следующем фрагменте кода показан сервлет-фильтр, который собирает значения времени отклика и пропускной способности в виде показателей гистограммы Prometheus:

```
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;

@WebFilter(urlPatterns = "/*")
public class MetricsCollectorFilter implements Filter {

    private Histogram requestDuration;

    @Override
    public void init(FilterConfig filterConfig) throws ServletException
{
        requestDuration = Histogram.build("request_duration_seconds",
                "Длительность HTTP-запросов в секундах")
                .buckets(0.1, 0.4, 1.0)
                .labelNames("request_uri")
                .register();
    }

    public void doFilter(ServletRequest req, ServletResponse res,
            FilterChain chain) throws IOException, ServletException {

        if (!(req instanceof HttpServletRequest)) {
            chain.doFilter(req, res);
            return;
        }

        String url = ((HttpServletRequest) req).getRequestURI();
        try (Histogram.Timer ignored = requestDuration
                .labels(url).startTimer()) {
            chain.doFilter(req, res);
        }
    }

    @Override
    public void destroy() {
        // ничего не делать
    }
}
```

Этот показатель регистрируется так же, как бизнес-показатель в предыдущем примере, и выводится через выходной формат Prometheus. На гистограмме показаны четыре временных отрезка, где указано время от 0,1, 0,4 или 1,0 с и выше. Эти конфигурации временных отрезков должны быть адаптированы к SLA.

Сервлет-фильтр активен для всех путей ресурсов и будет собирать статистику, соответствующую каждому из них.

Журналирование и отслеживание

Исторически сложилось так, что журналированию придавали очень большое значение в корпоративных приложениях. Нам встречалось множество фреймворков журналирования и предлагались всевозможные рекомендованные методы по разумной реализации журналов.

Журналирование обычно используется для отладки, отслеживания, ведения журналов, мониторинга и вывода ошибок. В журналы помещается вся информация, которую разработчики считают важной для себя, но не для пользователей. Почти всегда журналирование означает запись в файлы.

Недостатки традиционного журналирования

Этот подход, слишком широко распространенный в корпоративных проектах, сопряжен с несколькими проблемами.

Производительность

При традиционном журналировании, особенно с большим количеством обращений к журналу, создается множество строковых объектов. Даже такие API, как *Slf4J*, целью которых является сокращение лишних операций по слиянию строк, дают высокую нагрузку на память. После использования все эти объекты должны быть собраны, что нагружает процессор.

Хранение событий журнала в виде строковых сообщений — это слишком многословное хранение информации. Выбор других форматов, в основном двоичных, позволяет резко сократить размеры сообщений, эффективнее расходовать память и обеспечивает более высокую пропускную способность.

Журнальные сообщения, которые хранятся в буфере или непосредственно на диске, необходимо синхронизировать с другими вызовами журнала. Синхронные системы журналирования в итоге приводят к тому, что файл записывается за один вызов. Все одновременные вызовы журналов должны быть синхронизированы, чтобы гарантировать, что регистрируемые события будут записаны в правильном порядке. Это создает проблему: синхронизация косвенно связывает функции, которые иначе были бы полностью независимыми. В результате уменьшается возможность распараллеливания независимых функций и ухудшается производительность. При записи в журнал большого количества сообщений возрастает вероятность блокировки потоков из-за синхронизации.

Еще одна проблема заключается в том, что фреймворки журналирования обычно не записывают журнальные сообщения прямо на диск, а вместо этого используют несколько уровней буферизации. Этот метод оптимизации вызывает определенные накладные расходы, которые также не улучшают ситуацию. Синхронные операции с файлом стоит выполнять с минимально возможными накладными расходами.

Файлы журналов, которые находятся в системе NFS, еще больше снижают общую производительность, так как операция записи дважды обращается к системе ввода/вывода ОС, причем задействованы и файловая система, и сетевые вызовы. Для управления файлами журналов и записи в них часто предпочитают сетевое хранилище, особенно во фреймворке управления контейнерами, который нуждается в записываемых томах.

В целом, опыт показывает, что журналирование наиболее сильно влияет на производительность приложения. В основном это связано с воздействием памяти на строковые сообщения журнала.

Уровни журнальных записей

Решения для журналирования включают в себя возможность определять важность журнальных записей путем введения журнальных уровней, таких как *отладка*, *информация*, *предупреждение* или *ошибка*. Разработчикам стоит каждый раз задавать себе вопрос: какой уровень журнала выбрать для конкретных вызовов?

Концепция нескольких уровней — безусловно, разумный прием, поскольку позволяет в условиях эксплуатации задействовать более высокий уровень журнала, чем во время разработки, чтобы не создавать слишком много записей в рабочей среде.

Проблема заключается в том, что на этапе эксплуатации в журнал обычно не заносится информация уровня отладки и ее нельзя получить, когда это необходимо. Если возникают ошибки, которые могут потребовать дополнительной информации, то получить ее будет невозможно, так как журнальные уровни отладки и трассировки, содержащие эту информацию, были отключены.

Выбор уровней журналов — это всегда компромисс определения того, какую информацию стоит включить в журнал. Отладку в процессе разработки лучше всего выполнять с помощью специальных инструментов отладки, подключаемых к работающему приложению, возможно, удаленно. В режиме эксплуатации журналы отладки и трассировки обычно недоступны и, следовательно, бесполезны.

Возможно, в свое время журнал разделили на несколько уровней из хороших побуждений, но практическое применение этого приема в условиях эксплуатации приносит небольшую пользу.

Формат журнала

Традиционные решения для журналирования задают конкретные схемы журналов, определяющие формат сообщений, которые будут заноситься в файл журнала. На приложение возлагается задача управлять созданием, развертыванием и форматированием файлов журналов, которые не имеют отношения к бизнес-логике.

Довольно много корпоративных приложений поставляются в комплекте со сторонними зависимостями для ведения журнала, которые реализуют эту функциональность, но не представляют особой ценности для бизнеса.

Еще одно решение, которое необходимо принять при разработке приложения, — это выбор текстового формата для ведения журнала. Существует компромисс: формат журнальной записи, который могут читать как люди, так и машины. Результат, как правило, является худшим вариантом для обеих сторон: строковые форматы для журналов неудобочитаемы и значительно снижают производительность системы.

Разумнее выбирать двоичные форматы, позволяющие хранить информацию в более плотном виде. А люди могли бы применять специальные инструменты для отображения сообщений.

Объемы данных

Расширенное журналирование создает огромное количество данных, которые хранятся в файлах журналов. В частности, файлы журналов, используемых для отладки и трассировки, разрастаются до больших размеров, становятся громоздкими и неудобными для синтаксического анализа.

Синтаксический анализ форматов журналов обычно создает нагрузку на систему, которой можно избежать. Информация, которая может иметь техническое значение, сначала сериализуется в определенном формате, а затем при проверке журналов снова разделяется.

Далее в этом разделе мы рассмотрим другие решения этой проблемы.

Запутанность

Подобно необоснованному обслуживанию исключений, регистрация приводит к запутыванию бизнес-логики в исходном коде. Это особенно характерно для шаблонов журналов, используемых во многих проектах.

Записи журнала занимают слишком много места в коде и, что особенно важно, требуют от программиста много внимания.

Некоторые технологии журналирования, такие как Slf4j, предоставляют функциональные возможности для форматирования читаемых строк без их немедленной конкатенации. Но тем не менее записи журналов создают лишние вызовы, не связанные с бизнес-логикой.

Это в меньшей степени относится к случаю, когда записи журнала отладки добавляются в сквозной компонент, такой как перехватчик. Однако в этих случаях журналирование ведется в основном для трассировки. В следующем подразделе мы увидим, что для этого есть более подходящие решения.

Проблемы приложений

Как мы видели в приложениях, поддерживающих 12-факторную концепцию, выбор файлов и форматов сообщений для журналов не входит в круг задач приложения. В частности, фреймворки журналирования, которые предоставляют более простые решения для ведения журналов, добавляют к приложению технически обоснованные сторонние зависимости, которые не представляют непосредственной ценности для бизнеса.

Если события или сообщения ценны для бизнеса, то следует отдать предпочтение другому решению. Далее будет показано, что использование для этих целей традиционных журналов — неверное решение.

Ошибочный выбор технологии

Традиционное журналирование и то, как оно применяется в большинстве корпоративных проектов, — неоптимальный выбор для задач, которые лучше решаются другими методами.

Вопрос в следующем: что именно программисты хотят журналировать? Может быть, это такие показатели, как текущее потребление ресурсов? Или информация, связанная с бизнесом, например *количество выпущенных автомобилей*? Следует ли журналировать отладочную и отслеживающую информацию, такую как *запрос объекта с UUID xy от приложения A, вызывающий затем приложение B*? Или возникающие исключения?

Внимательные читатели могли заметить, что в большинстве случаев вместо традиционных журналов гораздо лучше использовать другие методы.

Если журналирование применяется для отладки или трассировки приложений, то журнал с уровнями трассировки или отладки не очень полезен. Если информация не будет доступна в среде эксплуатации, то она не поможет воспроизвести потенциальные ошибки. Однако журналирование большого количества событий отладки и трассировки в условиях эксплуатации повлияет на скорость реакции приложения из-за большого количества операций дискового ввода/вывода, синхронизации и потребления памяти. Отладка ошибок, связанных с параллелизмом, может даже привести к противоположному результату из-за измененной последовательности выполнения.

Для отладки гораздо лучше использовать во время разработки реальные функции отладчика, такие как IDE, которые подключаются к работающему приложению. Журналирование для целей бизнеса лучше делать с помощью специальных решений, которые будут описаны далее в этой главе. Журнальные сообщения в обычном текстовом формате, безусловно, не являются идеальным решением. Выбранная технология должна сводить к минимуму влияние на производительность приложения.

Еще один метод решения задач журналирования заключается в том, чтобы применять регистрацию событий. При этом события предметной области становятся частью основной модели приложения.

Журналирование, обоснованное целями бизнеса, также должно быть частью бизнес-сценария и реализовываться с использованием соответствующего решения. Как мы увидим в следующем разделе, есть более подходящие решения трассировки, требующие меньшего синтаксического анализа и слабее влияющие на производительность. Трассировочные решения также поддерживают консолидацию информации и запросы через микросервисы.

Информация мониторинга, которая хранится в сообщениях журнала, лучше управляется правильно выбранными решениями мониторинга. Такой подход

не только намного быстрее — это более эффективный способ вывода информации в виде правильных структур данных. Данные мониторинга и возможные решения проиллюстрированы на примерах ранее в этой главе.

Журналирование традиционно используется и для вывода исключений и ошибок, которые не могут быть обработаны в приложении другими способами. Это, пожалуй, единственное разумное применение журнала. Как и другие показатели, которые могут помочь обнаружить ошибку, например счетчики ошибок на границе системы, записанное в журнал исключение может помочь программистам в исследовании ошибок.

Однако ошибки и исключения следует заносить в журнал, только если они действительно касаются приложения и могут быть устранены программистами. При наличии решений мониторинга и предупреждения необходимость изучения журналов должна указывать на серьезную проблему с приложением.

Журналирование в мире контейнеров

Один из принципов 12-факторных приложений — рассматривать журналирование как поток событий. Сюда входит мысль о том, что обработка файлов журналов не должна относиться к кругу задач корпоративного приложения. Журнальные события должны просто выводиться на стандартный выход процесса.

Среда выполнения приложения объединяет и обрабатывает потоки журналов. Существуют решения для унифицированного доступа ко всем участвующим приложениям, которые могут быть развернуты в среде. Обработкой потоков журнала занимается среда выполнения, в которой развернуто приложение. Приложение *fluentd*, которое является частью Cloud Native Computing Foundation, унифицирует доступ к событиям журнала в распределенной среде.

Разработчики приложений должны обрабатывать используемую технологию журналирования как можно проще. Контейнер приложений сконфигурирован для вывода всех событий журнала сервера и приложений на стандартный выход. Такой метод упрощает задачи для разработчиков корпоративных приложений и позволяет им уделять больше внимания решению актуальных корпоративных проблем.

Как мы видели, информации, которую разработчикам приложений имеет смысл заносить в традиционный журнал, осталось совсем мало. Мониторинг, журналирование или решения трассировки, а также регистрация событий позволяют гораздо эффективнее решать эти задачи.

Когда события выводятся на стандартный выход без сложной обработки файлов журналов, исчезает необходимость в сложном фреймворке журналирования. Это соответствует идее приложений с нулевой зависимостью и позволяет разработчикам сосредоточиться на проблемах бизнеса.

Поэтому рекомендуется избегать сторонних фреймворков журналирования, а также записи в традиционные файлы журналов. Это избавляет от необходимости

управлять обменом журналов, форматами журнальных записей, уровнями и зависимостями фреймворков, а также конфигурацией.

Однако разработчикам корпоративных приложений это может показаться противоречивым.

Прямолинейный 12-факторный способ вывода журнальных событий на стандартный выход использует возможности стандартного вывода Java через System.out и System.err. При этом происходит непосредственная запись на синхронный выход без лишних уровней буферизации.

Важно отметить, что при этом данные выводиться не будут. Введенная синхронизация соединяет части приложения, которые без нее являются независимыми. Если выход процесса захватывается и выводится видеокартой, то производительность будет еще ниже.

Вывод записей журнала в консоль предназначен только для вывода ошибок, которые, как показывает имя типа Java, являются исключениями. Во всех остальных случаях инженеры должны задать себе вопрос, хотят ли они вывести эту информацию в первую очередь или же есть другие, более подходящие решения. Таким образом, журналированные ошибки должны указывать на фатальную проблему, требующую вмешательства инженеров. Вывод таких журнальных сообщений не должен происходить в условиях эксплуатации — в случае фатальных ошибок производительность может упасть ниже всяких пределов.

Для вывода информации о фатальных ошибках в приложении Java EE можно использовать функции CDI, а также функциональные интерфейсы Java SE 8, обеспечивающие единую функциональность журналирования:

```
public class LoggerExposer {

    @Produces
    public Consumer<Throwable> fatalErrorConsumer() {
        return Throwable::printStackTrace;
    }
}
```

Регистратор `Consumer<Throwable>` затем внедряется в другие компоненты и регистрируется с помощью метода `accept()` типа `Consumer`. Если требуется более удобный для чтения интерфейс, то может быть создан такой фасад регистратора, который внедряется с помощью `@Inject`:

```
public class ErrorLogger {

    public void fatal(Throwable throwable) {
        throwable.printStackTrace();
    }
}
```

Разработчикам корпоративных приложений такой подход может показаться противоречивым, особенно идея журналирования без фреймворка. Задействование сложных фреймворков журналирования, предназначенных для перенаправления вывода на то же стандартное устройство, дает дополнительную нагрузку

на ресурсы и в итоге приводит к тому же результату. Некоторые разработчики предпочитают на этом этапе использовать журналирование JDK.

Однако предлагать сложные интерфейсы журналирования и таким образом давать разработчикам приложений возможность выводить любые виды информации, особенно легко читаемые строки, непродуктивно. Именно поэтому приведенные примеры кода позволяют выводить только такие типы информации, которые обычно выводятся при возникновении исключений, в случае фатальных ошибок.

Учитывайте следующие особенности.

- Нужно избегать традиционного журналирования, предпочитая ему более подходящие решения.
- В журнал следует заносить только фатальные ошибки, которые являются исключением и в идеале никогда не должны возникать.
- В случае контейнерных приложений рекомендуется выводить журналы событий на стандартный выход.
- Журналирование и интерфейсы в приложениях должны быть как можно более простыми, чтобы разработчики не использовали их без особой нужды.

Журналирование

Если журналирование является частью бизнес-логики, есть более эффективные пути, чем использование фреймворков. Журналирование бывает нужно в системах для аудита или торговли.

Если бизнес-логика требует журналирования, то его следует рассматривать как необходимость для бизнеса. Существуют технологии журналирования, которые синхронно сохраняют требуемую информацию с более высокой плотностью и меньшей задержкой, чем при традиционном журналировании. Примером такого решения является *Chronicle Queue*, которое позволяет сохранять сообщения с высокой пропускной способностью и низкой задержкой.

Предметная область приложения может моделировать информацию в виде событий предметной области и напрямую переносить ее в решение для журналирования. Как уже отмечалось, другой способ заключается в том, чтобы приложение базировалось на модели регистрации событий. Тогда информация аудита является частью модели приложения.

Трассировка

Трассировка применяется для воспроизведения конкретных сценариев и потоков запросов. Это, очевидно, полезно для повторения сложных процессов приложения, но еще более полезно, когда задействованы несколько приложений и экземпляров.

Однако важно отметить, что, как и в случае с журналированием, основанием для использования системы трассировки должны быть требования бизнеса, а не технические условия.

Трассировка — плохой способ отладки или наблюдения за производительностью системы. Она оказывает определенное влияние на производительность и не очень помогает в решении проблем с ней. Взаимозависимые, распределенные приложения, которые нуждаются в оптимизации производительности, должны лишь выводить информацию о качестве обслуживания, например время отклика. Технологии выборочных исследований позволяют собирать достаточно информации, которая может указывать на проблемы производительности в приложениях.

Однако рассмотрим случай, когда трассировка нужна по соображениям бизнеса, чтобы отслеживать задействованные компоненты и системы.

На рис. 9.5 показана трассировка запроса, требующего участия нескольких экземпляров приложения и компонентов.

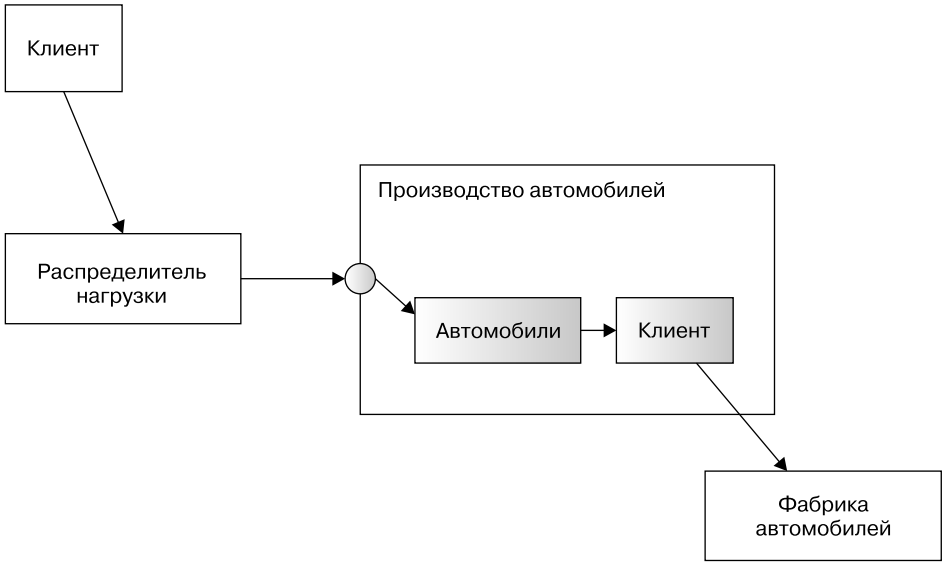

Рис. 9.5

Трассировку также можно отображать на временной шкале, чтобы показать синхронные вызовы, как на рис. 9.6.

Трассировка включает в себя информацию о том, какие приложения или компоненты приложений были задействованы и как долго выполнялись отдельные вызовы.

Традиционно для этого использовались файлы журналов, в которые заносилось время начала и завершения каждого вызова метода или компонента, а также идентификатор корреляции, такой как идентификатор потока. В журналы можно включать идентификаторы корреляции, которые берутся из общего исходного

запроса и затем применяются повторно и регистрируются в последующих приложениях. Так появляются результаты трассировки, охватывающие несколько приложений.

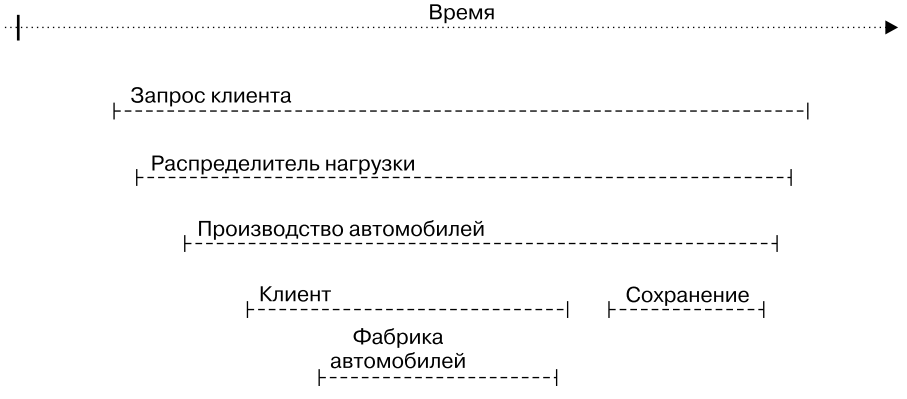

Рис. 9.6

В случае журналирования информация трассировки накапливается в нескольких файлах журнала, в частности при использовании таких решений, как стек ELK. Журналы трассировки обычно реализуются в виде сквозных задач — например, с помощью фильтров журналирования и перехватчиков, чтобы не усложнять код.

Однако задействовать файлы журналов для трассировки нецелесообразно. Даже если корпоративные приложения испытывают умеренную нагрузку, они все равно вносят много записей в журнал, существующий в виде файлов. При каждом запросе создается несколько журнальных записей.

Файловый ввод-вывод и необходимая сериализация данных для журнала, как правило, слишком тяжелы для этого метода и сильно влияют на производительность. При трассировке с записью в журнальные файлы создается много данных, которые впоследствии необходимо обрабатывать синтаксическим анализатором.

Существуют другие решения для трассировки, которые гораздо лучше справляются с этими задачами.

Трассировка в современном мире. В последние годы возникло множество решений для трассировки, цель которых — свести к минимуму ее влияние на производительность системы.

Стандартной, независимой от поставщиков технологией трассировки является *OpenTracing*, входящая в состав Cloud Native Computing Foundation. Она определяет понятия и семантику результатов трассировки и поддерживает трассировку в распределенных приложениях. Реализована в виде нескольких технологий трассировки, таких как *Zipkin*, *Jaeger* и *Hawkular*.

Иерархический результат трассировки состоит из нескольких интервалов, подобных показанным на предыдущих рисунках. Интервал может быть дочерним по отношению к другому интервалу или следовать за ним.

В предыдущем примере интервал «производство автомобилей» является дочерним для интервала «распределитель нагрузки», а интервал «сохранение» идет после интервала «клиент», так как их вызовы происходят последовательно.

В API OpenTracing описание интервала включает в себя временной интервал, имя операции, контекстную информацию, а также дополнительные наборы тегов и журналов. Названия операций и теги подобны именам и меткам показателей в Prometheus, описанным ранее в разделе «Введение в Prometheus». В журналах хранится такая информация, как сообщения интервала.

Примером отдельного интервала является `createCar` с тегами `color=RED` и `engine=DIESEL`, а также полем `Car successfully created` журнала `message`.

В следующем фрагменте кода приводится пример использования OpenTracing Java API для приложения «производство автомобилей». Он поддерживает конструкцию Java try-with-resource:

```
import io.opentracing.ActiveSpan;
import io.opentracing.Tracer;

@Stateless
public class CarManufacturer {

    @Inject
    Tracer tracer;

    public Car manufactureCar(Specification spec) {
        try (ActiveSpan span = tracer.buildSpan("createCar")
                .withTag("color", spec.getColor().name())
                .withTag("engine", spec.getEngine().name())
                .startActive()) {

            // выполнение бизнес-логики

            span.log("Car successfully created");
        }
    }
}
```

Созданный интервал активно запускается и добавляется как дочерний в потенциально уже существующий родительский интервал. `Tracer` генерируется CDI-генератором, который зависит от конкретной реализации OpenTracing.

Очевидно, что такой подход сильно усложняет код и его следует перенести в сквозные компоненты, такие как перехватчики. Сборка перехватчиков трассировки может декорировать методы и извлекать информацию об именах и параметрах метода.

В зависимости от информации, которую желательно включить в интервалы трассировки, привязка перехватчика может быть расширена и в нее может быть включена дополнительная информация, такая как имя операции.

В следующем фрагменте кода представлен бизнес-метод, декорированный с привязкой перехватчика, который добавляет трассировку рациональным

способом (реализацию перехватчика предлагаю выполнить читателю в качестве упражнения):

```
@Stateless
public class CarManufacturer {

    …

    @Traced(operation = "createCar")
    public Car manufactureCar(Specification spec) {
        // выполнение бизнес-логики
    }
}
```

Информация трассировки переносится в последующие приложения через контексты и носители интервалов. Они также позволяют задействованным приложениям добавлять свои данные трассировки.

Собранные данные могут быть извлечены с помощью используемой реализации OpenTracing. Существуют реализации фильтров и перехватчиков, доступные для таких технологий, как ресурсы и клиенты JAX-RS, которые прозрачно добавляют в вызовы всю необходимую отладочную информацию, например, с помощью заголовков HTTP.

Этот способ трассировки меньше влияет на производительность системы, чем традиционное журналирование. Он определяет точные операции и системы, которые реализуют поток бизнес-логики. Однако, как уже отмечалось, применение решения для трассировки должно быть обосновано требованиями бизнеса.

Типичные проблемы производительности

Проблемы производительности проявляются в виде типичных признаков, таких как замедленное или постепенно замедляющееся время отклика, задержки или даже полностью недоступные сервисы. На последнее обстоятельство указывает частота ошибок.

Когда возникают проблемы с производительностью, следует задать себе несколько вопросов. Какой ресурс на самом деле является ограничивающим? Где находится узкое место? Что является источником проблемы? Как было показано ранее, для того чтобы найти причину, инженерам рекомендуется провести расследование, в котором учесть общую ситуацию, включая аппаратные средства и операционные системы. Не должно быть никаких догадок и преждевременных решений.

Проблемы с производительностью могут иметь огромное количество первопричин. Большинство из них вызвано ошибками в коде или конфигурации, а не тем, что рабочая нагрузка действительно слишком велика для доступных ресурсов. Современные серверы приложений способны справляться с большой нагрузкой, прежде чем производительность станет проблемой.

Однако опыт показывает, что существуют типичные причины проблем с производительностью. Далее мы покажем самые серьезные из них. Инженерам следует тщательно расследовать эти проблемы, не полагаясь на то, что принято считать рекомендованными методами разработки и не предпринимая преждевременные оптимизации.

Журналирование и потребление памяти

Традиционные методы журналирования, такие как запись журнальных сообщений в виде строк в файлы, — наиболее распространенная причина низкой производительности. В этой главе уже описаны эти проблемы и рекомендации по их решению.

Основной причиной низкой производительности является создание множества строковых объектов и, следовательно, большой расход памяти. Высокое потребление памяти создает серьезную проблему с производительностью. Это может быть вызвано не только журналированием, но и большим расходом памяти при кэшировании, утечками памяти или созданием большого количества объектов.

Поскольку JVM управляет сборкой мусора в памяти, большой расход памяти приводит к частому запуску сборщика мусора, который пытается освободить неиспользуемую память. При сборке мусора задействуется процессор. Данная ситуация не устраняется одним запуском сборщика, так как он вызывает последующие запуски GC и, соответственно, дает высокую нагрузку на процессор. Это происходит в тех случаях, когда память не освобождается из-за утечек памяти или высокой рабочей нагрузки с высоким уровнем потребления. Даже если система не даст сбой `OutOfMemoryError`, перегрузка процессора может привести к остановке приложения.

В исследовании этих проблем могут помочь журналы сбора мусора, дампы кучи и измерения. Инструменты JMX дают представление о распределении памяти и возможных проблемных точках.

Если бизнес-логика реализована простым и лаконичным способом с использованием короткоживущих объектов, то проблемы с памятью гораздо менее вероятны.

Преждевременная оптимизация

Разработчики корпоративных проектов часто пытаются преждевременно оптимизировать приложения, не выполнив надлежащую проверку. Примерами этого являются использование кэширования, настройка пулов и поведения сервера приложений без выборочного исследования до и после настройки.

Настоятельно рекомендуется не проводить эти оптимизации до того, как будет выявлена проблема с производительностью. Прежде чем изменять конфигурацию системы, необходимо провести правильные выборочные исследования производительности и измерения в условиях эксплуатации, а также проанализировать ограничивающий ресурс.

В подавляющем большинстве случаев достаточно согласовать конфигурацию. Это справедливо как для среды выполнения JVM, так и для сервера приложений. Если разработчики используют простой метод Java EE со стандартной конфигурацией сервера приложений, то они вряд ли столкнутся с проблемами преждевременной оптимизации.

И только в том случае, если, судя по техническим показателям, выбранной конфигурации недостаточно для рабочей нагрузки, необходимо внести изменения. Кроме того, инженеры должны периодически проверять необходимость сделаных изменений. Технологические изменения и оптимизация, позволившие устранить проблему в предыдущих версиях рабочей среды, со временем могут стать не лучшим решением.

Концепция программирования по соглашениям и первоначальный выбор стандартной конфигурации также требуют наименьшего количества начальных усилий.

И снова напомню: как показывает опыт, многие проблемы возникают из-за преждевременного внесения изменений без надлежащей предварительной проверки.

Реляционные базы данных

Обычно при недостаточной производительности козлами отпущения назначают реляционные базы данных. Как правило, серверы приложений развертываются в нескольких экземплярах и все подключаются к одному экземпляру базы данных. Это необходимо для согласованности в соответствии с теоремой CAP.

База данных как единая точка ответственности, или точка проблемы, по определению становится первым кандидатом на роль узкого места. Тем не менее инженеры должны сделать соответствующие измерения, чтобы подтвердить это предположение.

Если, согласно показателям, отклик базы данных медленнее допустимого, то прежде всего, как обычно, нужно исследовать первопричину этой ситуации. Если причиной медленного отклика является запрос базы данных, то инженерам рекомендуется изучить выполняемые запросы. Может быть, загружается много данных? Если да, то все ли эти данные нужны, или они впоследствии будут отфильтрованы и сокращены? Бывает так, что запросы к базе данных загружают больше данных, чем требуется.

То же касается бизнес-логики, особенно извлечения данных: все ли они нужны? В этих случаях могут оказаться эффективными более точные запросы к базе данных с предварительной фильтрацией результатов или ограничениями по размеру, такими как разбиение на страницы.

Базы данных очень хорошо справляются с объединением и фильтрацией данных. Выполнять более сложные запросы непосредственно в экземпляре базы данных обычно эффективнее, чем загружать все данные в память приложения и выполнять запрос там. На Java можно создавать сложные вложенные SQL-запросы и выполнять их в базе данных. Однако в корпоративных приложениях

следует избегать определения запросов бизнес-логики непосредственно в базе данных с применением хранимых процедур. Бизнес-логика должна находиться в приложении.

Еще одна типичная ошибка конфигурации — пренебрежение индексированием столбцов базы данных, которые часто используются в запросах. В проектах часто удается повысить общую производительность в несколько раз, всего лишь создав правильные индексы.

В целом, аналитические измерения конкретных бизнес-сценариев обычно дают хорошее представление о возможных причинах проблемы.

В некоторых сценариях запросы, обновляющие данные, часто приводят к ошибкам оптимистической блокировки. Это происходит из-за одновременного обновления сущностей предметной области. Оптимистическая блокировка — это скорее проблема бизнес-сценария, чем техническая. На возникновение подобных проблем указывает частота ошибок сервиса.

Если в бизнес-сценарии требуется, чтобы сущности часто одновременно изменялись, программисты могут рассмотреть возможность изменения функциональности на модель, управляемую событиями. Или же, как было показано ранее, при регистрации событий и в событийно-ориентированной архитектуре можно избавиться от этой ситуации, введя принцип итоговой согласованности.

Если проблемы с производительностью возникают только из-за рабочей нагрузки и конкурентного доступа, то нужна другая модель данных, такая как событийно-ориентированная архитектура, реализованная с помощью CQRS. Однако, как правило, эта ситуация разрешается другими способами. Подавляющее большинство корпоративных приложений довольно хорошо масштабируется с использованием реляционных баз данных.

Коммуникация

Большинство проблем с производительностью, связанных с коммуникацией, относятся к сфере синхронной связи. Проблемы в этой области возникают главным образом из-за отсутствия задержек и приводят к тому, что вызовы клиентов постоянно блокируются и наблюдается взаимоблокировка. Это происходит, если не настроены задержки на стороне клиента и вызываемая система недоступна.

Менее критическая, но тоже неудачная ситуация возникает, если выбраны слишком большие задержки. Это заставляет системы ожидать слишком долго, замедляя процессы и блокируя потоки.

Правильный выбор задержек для клиентских вызовов, как было описано ранее, — это простое, но эффективное решение этой проблемы.

Большое время отклика и низкая пропускная способность могут иметь несколько причин. Для того чтобы узнать, на что затрачивается время, нужно выполнить анализ производительности.

Вероятно образование и других узких мест, таких как объемы полезной нагрузки. Размер полезной нагрузки может зависеть от того, передаются ли данные в виде обычного текста или в двоичном формате. Сериализация, для которой используются несовершенные алгоритмы или технология, также может уменьшать скорость реакции. Однако все эти проблемы обычно не слишком важны, если нагрузка на приложение невысока.

Если требуется выполнить несколько синхронных вызовов, то это следует делать по возможности параллельно, используя потоки, управляемые контейнерами, например предоставляемые службой запланированных исполнителей. Это позволяет избавить приложение от излишнего ожидания.

В общем случае следует избегать бизнес-сценариев с применением нескольких транзакционных систем, таких как базы данных с распределенными транзакциями. Как уже говорилось, распределенные транзакции не масштабируются. Вместо этого бизнес-сценарий должен эффективно выполняться в асинхронном режиме.

Потоки и пулы

Для того чтобы многократно применять потоки и соединения, контейнеры приложений объединяют их в пулы. Тогда запрошенные потоки не обязательно создавать заново, их можно повторно брать из пула.

Пулы создаются для управления нагрузкой на определенные части системы. Выбор правильного размера пула обеспечивает хорошую нагрузку системы, но предотвращает перегрузку. Это связано с тем, что пустые пулы приведут к приостановлению или отклонению запросов. Это значит, что все потоки и соединения этого пула уже используются.

Шаблон переборки предотвращает влияние разных частей системы друг на друга, создавая выделенные пулы потоков. В этом случае нехватка ресурсов может быть ограничена только конкретной проблемной функцией. В некоторых случаях причиной проблем могут быть устаревшие системы. Переборки, реализованные в виде выделенных пулов потоков, и правильный выбор задержек помогают сохранить работоспособность приложения.

Причиной появления пустых пулов может быть либо слишком высокая текущая нагрузка на данный пул, либо гораздо более длительное, чем ожидалось, использование ресурсов. В любом случае рекомендуется не просто увеличить размер соответствующего пула, а исследовать источник проблем. Описанные методы исследования, а также изучение JMX и дампов потоков помогут вам найти узкие места, а также потенциальные ошибки программирования, такие как взаимоблокировки, неверно выбранные задержки и утечки ресурсов. Недостаток потоков в пуле из-за высокой рабочей нагрузки возникает гораздо реже.

Размеры и конфигурация пула выбираются в контейнере приложения. Для этого инженеры должны надлежащим образом измерить производительность в условиях эксплуатации до и после перенастройки сервера.

Тестирование производительности

Проблема тестирования производительности заключается в том, что эти тесты выполняются в имитируемой среде.

Имитируемые среды отлично подходят для других видов тестирования, например с помощью системных тестов, поскольку некоторые показатели можно абстрагировать. Например, макетирование серверов позволяет имитировать поведение, аналогичное условиям эксплуатации.

Однако, в отличие от функциональных тестов, проверка скорости реакции системы требует учета всего, что происходит в условиях эксплуатации. В конце концов, приложения работают на реальном оборудовании, поэтому на производительность приложения влияют аппаратное обеспечение, а также общая ситуация. Производительность системы в имитируемых средах никогда не будет такой, как в среде эксплуатации. Поэтому тесты производительности — это ненадежный способ поиска узких мест.

Есть множество сценариев, в которых приложение может работать намного лучше, чем в тестах производительности, в зависимости от всех непосредственных и неизбежных воздействий. Например, JVM HotSpot лучше работает при высокой нагрузке. Поэтому исследование ограничений производительности следует выполнять только в среде эксплуатации. Как было показано ранее, процессы исследования jPDM, а также методы и инструменты выборочных исследований позволяют найти узкое место, если использовать их в условиях эксплуатации.

Тесты производительности и стресс-тесты помогают найти очевидные ошибки кода или конфигурации, такие как утечка ресурсов, грубые ошибки в конфигурации, отсутствие задержек и взаимные блокировки. Они будут обнаружены до развертывания в среде эксплуатации. Тесты производительности позволяют также отследить тенденции изменения производительности с течением времени и предупредить инженеров о возможном падении скорости реакции. Однако они могут свидетельствовать только о потенциальных проблемах, так что не следует делать преждевременные выводы на их основе.

Выполнять тесты производительности и стресс-тесты имеет смысл только для всей сети взаимозависимых приложений с учетом всех зависимостей и взаимного влияния производительности всех задействованных систем и баз данных. Среда должна быть максимально подобной условиям эксплуатации.

Но даже в этом случае результат не будет таким же, как в среде эксплуатации. Очень важно, чтобы инженеры это понимали. Поэтому оптимизация производительности по результатам тестов никогда не может быть точной.

Для точной настройки производительности важно применять выборочные исследования и изучение результатов в среде эксплуатации. Поддержка методов непрерывной поставки кода помогает быстро перенести изменения конфигурации в среду эксплуатации. Затем инженеры могут использовать выборку и методы исследования производительности, чтобы узнать, улучшилась ли ситуация. И снова напомню: необходимо учитывать состояние всей системы. Простая настройка одного приложения без учета остальной системы может иметь негативные последствия для общего сценария.

Резюме

Показатели, связанные с бизнесом, могут обеспечить полезную информацию о корпоративном приложении. Они являются частью бизнес-сценария и должны рассматриваться как таковые. На бизнес-показатели в конечном итоге влияют другие, технические показатели. Поэтому их также следует контролировать.

Согласно теории ограничений, может существовать одно или несколько ограничивающих условий, мешающих бесконечному увеличению пропускной способности системы. Для того чтобы повысить производительность приложения, необходимо устранить ограничивающие условия. jPDM помогает идентифицировать такие условия, сначала найдя доминирующего потребителя ресурсов процессора, а затем используя соответствующие инструменты для дальнейшего изучения проблем производительности. Стоит исследовать потенциальные узкие места, следуя этому процессу, который учитывает общую ситуацию, а не пользоваться методом научного тыка.

Вместо высокочастотного мониторинга инженерам рекомендуется делать выборки технических показателей с низкой частотой и запрашивать, вычислять и исследовать данные вне системы. Это существенно слабее влияет на производительность приложения. Распределенные приложения должны соответствовать требованиям SLA. Метод замедленной обратной реакции, а также шаблон переборки помогают строить устойчивые корпоративные системы с коротким временем отклика.

По ряду причин следует избегать традиционного журналирования, особенно негативно влияющего на производительность. В корпоративных приложениях рекомендуется выводить события журнала только в случае фатальных и непредвиденных ошибок, которые передаются на стандартный выход максимально простым способом. Для всех других причин, таких как отладка, трассировка, ведение журнала или мониторинг, существуют более подходящие решения.

Тесты производительности и стресс-тесты, выполняемые в имитируемых средах, могут использоваться для обнаружения очевидных ошибок в приложении. Среда должна быть как можно более близкой к эксплуатационной и включать все приложения и базы данных. В отношении всех остальных рассуждений, особенно предположений об ожидаемой производительности приложения, узких местах и оптимизации, тесты производительности не дают надежных результатов и могут даже привести к ошибочным выводам.

Следующая глава посвящена теме безопасности приложений.

10 Безопасность

До сих пор при рассмотрении большинства тем в этой книге мы не затрагивали вопросов безопасности. Ее часто упускают из виду, и в некоторых реальных проектах вспоминают о ней только тогда, когда уже слишком поздно.

Разработчики и руководители проектов смотрят на безопасность как на неизбежное зло, а не как на что-то, что может быть очень полезным для бизнеса. Однако все заинтересованные стороны должны считать именно так.

В эпоху облачных и распределенных приложений требования к безопасности довольно сильно изменились. В этой главе рассказывается, как было в прошлом, а также каковы сегодняшние требования. Рассмотрим следующие способы обеспечения безопасности с применением современной Java EE.

- Уроки безопасности, извлеченные из прошлого.
- Принципы корпоративной безопасности.
- Современные решения для обеспечения безопасности.
- Как обеспечить безопасность с использованием современной Java EE.

Уроки прошлого

В современном мире информационная безопасность — очень важный фактор. Большинство людей осознало, что при неправильном применении информационные технологии способны причинить большой вред.

За последние полвека использования компьютеров человечество многому научилось в смысле безопасности, и не только в области корпоративного программного обеспечения.

Рассмотрим здесь несколько уроков, извлеченных из прошлого опыта разработки корпоративных приложений. Раньше самыми важными проблемами безопасности были шифрование и управление регистрационными данными.

Шифрование и электронные подписи — это беспрецедентно безопасный способ хранения секретных данных, если применять его правильно. Безопасность зависит исключительно от используемых алгоритмов и длины ключей.

Предлагалось довольно много алгоритмов шифрования и хеширования, и все они оказались недостаточно безопасными. Примерами таких алгоритмов являются *DES*, а также часто используемый алгоритм хеширования *MD5*. На момент написания книги достаточно защищенным считался *AES* с длиной ключа 192 или 256 бит. Для алгоритма хеширования рекомендуется *SHA-2* или *SHA-3* с длиной ключа по меньшей мере 256 бит.

Если регистрационные данные пользователя являются частью приложения, то они не должны храниться в обычном текстовом формате. В прошлом было обнаружено слишком много брешей в системе безопасности, особенно в отношении баз данных, в которых хранились пароли. Кроме того, не рекомендуется просто хешировать пароли, без использования соли.

В общем случае очень важно, чтобы разработчики корпоративных приложений не реализовывали функции безопасности самостоятельно, если этого можно избежать. В компаниях мыслили так: нужно создавать собственные реализации систем безопасности, которые нигде не применялись и, следовательно, должны обеспечивать безопасность, так как неизвестны всем прочим. Однако результат оказывался противоположным: если только не привлекались эксперты по безопасности, создавались не совсем безопасные решения.

Подавляющее большинство условий корпоративной безопасности не требуют специальных самостоятельно написанных реализаций. Корпоративные фреймворки и их реализации поставляются с готовой функциональностью, хорошо протестированной в многочисленных бизнес-сценариях. Далее в этой главе мы рассмотрим эти API для Java Enterprise.

Если приложение требует нестандартного шифрования, то следует использовать реализации, предоставляемые средой выполнения или сторонними разработчиками. По этой причине платформа Java включает в себя расширение *Java Cryptography Extension (JCE)*, обеспечивающее реализацию современных алгоритмов шифрования и хеширования.

В целом приложения должны обрабатывать и хранить защищенную информацию, только когда это абсолютно необходимо для бизнес-сценариев. В частности, существуют способы избежать хранения учетных данных пользователей для аутентификации и авторизации в нескольких системах.

Безопасность в современном мире

Распространение приложений приводит к увеличению спроса на защищенную связь. Требуется обеспечить целостность передаваемой информации. Также люди понимают необходимость шифрования, особенно когда речь заходит об обмене данными.

Какие возможности есть у инженеров в современном корпоративном мире? Какими принципами они должны руководствоваться при обеспечении безопасности?

Принципы обеспечения безопасности

Есть ряд базовых принципов, которых необходимо придерживаться при обеспечении безопасности корпоративных приложений. Приведенный далее список не является исчерпывающим, но он позволит вам составить общее представление о них.

Шифрование связи

Прежде всего важно отметить, что внешние коммуникации, осуществляемые через Интернет, должны быть зашифрованы. Обычно это делается это через TLS с использованием доверенных сертификатов. Это возможно для HTTP, а также других протоколов связи.

Аутентификация сертификатов должна проверяться реализацией приложения во время выполнения. Они должны подтверждаться доверенным внутренним или внешним центром сертификации.

Принимать любые сертификаты в приложении без проверки небезопасно, и этого следует избегать как в условиях эксплуатации, так и в других средах. Это означает, что для связи предоставляются и используются только надлежащим образом подписанные сертификаты.

Делегирование обеспечения безопасности

В настоящее время подход к хранению информации о пользователях заключается в делегировании полномочий и авторизации у поставщиков услуг по обеспечению безопасности в той степени, в которой это возможно. Это означает, что корпоративное приложение не хранит информацию, касающуюся безопасности, а запрашивает ее у доверенного стороннего поставщика услуг по безопасности.

Это особенно интересно в распределенных средах, где несколько приложений предлагают потенциальные конечные точки доступа к внешнему миру. Безопасная передача информации сводится в единую точку ответственности.

Задачи безопасности обычно выходят за пределы основной бизнес-логики. Приложение делегирует доверенному поставщику услуг по безопасности возможность проверить безопасность пользовательских запросов. Поставщик услуг по безопасности действует как единая безопасная точка ответственности. Этот подход реализован в децентрализованных протоколах безопасности, таких как *OauthorOpenID*.

Делегирование ответственности доверенному поставщику услуг по безопасности исключает необходимость передавать пароли в корпоративных системах. Идентификация пользователей происходит непосредственно у поставщиков услуг по безопасности. Приложениям, которые требуют секретной информации о пользователе, предоставляются маркеры сеанса, которые сами по себе не содержат конфиденциальной информации.

Однако этот принцип в основном ориентирован на коммуникацию, где под пользователями приложений подразумеваются люди.

Правильная обработка пользовательских учетных данных

Если по какой-либо причине приложение само управляет аутентификацией пользователей, то оно ни в коем случае не должно постоянно хранить пароли и маркеры сеанса в виде открытого текста. Это создает серьезную угрозу безопасности. Даже если приложение или база данных достаточно хорошо защищены от доступа из внешнего мира, важно также защитить учетные данные от внутренних утечек.

Пароли, которыми необходимо управлять в приложении, должны храниться только хешированными с помощью соответствующих алгоритмов и методов, таких как использование *соли*. Это предотвращает любую злонамеренную атаку как изнутри, так и извне организации. Рекомендуется проконсультироваться по этому вопросу с организациями по обеспечению безопасности, такими как *Open Web Application Security Project (OWASP)*. Они дают актуальные рекомендации по методам и алгоритмам безопасности.

Избегайте хранения учетных данных в системах контроля версий

По той же причине, по которой не следует относиться небрежно к секретным учетным данным, программисты не должны хранить учетные данные в виде открытого текста в репозитории проекта под управлением системы контроля версий. Это создает угрозу безопасности, даже если хранилище находится внутри компании. Учетные данные будут постоянно видимыми в истории хранилища.

Как было показано в главе 5, в облачных средах существуют функции, которые внедряют в приложения секретные значения конфигурации. Эти функции могут задействоваться для предоставления секретных учетных данных, конфигурируемых извне.

Используйте тесты

Механизмы обеспечения безопасности, включенные в состав приложения, должны быть тщательно протестированы на уровне системных тестов. Любые включенные аутентификация и авторизация должны быть проверены как часть конвейера непрерывной поставки. Это означает, что нужно проверять функциональность в автоматизированных тестах не один раз, а постоянно, после любого изменения программного обеспечения.

Особенно важно, чтобы тестирование безопасности включало в себя отрицательные тесты. Например, тест должен показывать, что неправильные учетные

данные или недостаточные разрешения не позволяют выполнять определенные функции приложения.

Возможности и решения

Рассмотрев ряд основных принципов безопасности, перейдем к возможным протоколам и решениям по обеспечению безопасности.

Шифрованная коммуникация

Зашифрованная коммуникация обычно означает, что связь шифруется методом *TLS-шифрования* в составе протокола связи на транспортном уровне. Сертификаты используются для шифрования и подписи сообщений. Разумеется, очень важно иметь возможность полагаться на сертификаты.

Компании часто работают с собственными *органами сертификации (certificate authorities, CA)* и предварительно устанавливают *корневой CA* на своих компьютерах и в программном обеспечении. Это определенно имеет смысл для внутренних сетей: уменьшаются накладные расходы и потенциальные затраты по сравнению с запросами сертификатов для всех внутренних служб от официальных органов сертификации.

Сертификаты, которые *пользуются общим доверием*, должны быть подписаны одним из официальных органов сертификации, которые предварительно устанавливаются в операционной системе или на платформе.

Шифрованная связь не аутентифицирует пользователей, если только не используются индивидуальные клиентские сертификаты. Она только закладывает основу для безопасной и надежной коммуникации.

Аутентификация на основе протокола

Некоторые протоколы связи имеют возможность аутентификации — например, у HTTP есть базовая, или дайджест-аутентификация. Эти функциональные возможности являются частью протокола связи и обычно хорошо поддерживаются инструментами и фреймворками.

Чаще всего они подразумевают, что сообщение уже надежно зашифровано, иначе информация станет доступной для тех, кто перехватит сообщение, — в этом случае он сможет его прочитать. Это важно помнить разработчикам приложений: аутентификация на основе протокола должна обеспечиваться через зашифрованное соединение.

Регистрационные данные для обеспечения безопасности на основе протокола обычно вставляются прямо в сообщение. Это упрощает клиентские вызовы, так как нет необходимости выполнять многоэтапную аутентификацию, как происходит, например, при обмене маркерами сеанса. Уже при первом клиентском вызове можно передавать информацию.

Децентрализованная безопасность

Другие методы, которые непосредственно не включают регистрационные данные в клиентские вызовы, сначала получают маркеры безопасности, а затем устанавливают фактическое соединение по предоставленному маркеру. Это шаг в направлении децентрализованного обеспечения безопасности.

Для того чтобы отделить безопасность от приложения, корпоративные системы могут включать в себя поставщиков удостоверений как центральную точку аутентификации или авторизации. Таким образом задачи обеспечения безопасности делегируются от приложения поставщику.

Поставщики удостоверений авторизируют третьи стороны, такие как корпоративные приложения, без прямого обмена регистрационными данными. Конечные пользователи перенаправляются к поставщикам удостоверений и не передают защищенную информацию корпоративному приложению. Третьи стороны получают информацию только в том случае, если доступ разрешен, посредством маркеров, которые они могут проверить.

Такая трехсторонняя аутентификация позволяет избежать переноса обязанностей по обеспечению безопасности в корпоративное приложение. Ответственность за проверку правильности информации, предоставленной пользователем, переходит к поставщику удостоверений.

Одним из примеров этого метода являются механизмы *системы единого входа (single sign on, SSO)*. Их часто применяют в крупных компаниях, чтобы пользователи аутентифицировались только один раз и эта информация затем использовалась во всех сервисах, защищенных SSO. Система SSO аутентифицирует пользователя и предоставляет необходимую информацию о нем соответствующим приложениям. Пользователям нужно только один раз войти в систему.

Другим вариантом является задействование протоколов делегирования децентрализованного доступа, таких как OAuth, OpenID и OpenID Connect. Они представляют собой трехсторонние процессы обеспечения безопасности для обмена конфиденциальной информацией между клиентами, сторонними приложениями и поставщиком удостоверений. Принцип их работы аналогичен механизмам единого входа. Однако эти протоколы позволяют пользователям решать, какое конкретное приложение получит их информацию. Вместо фактических регистрационных данных приложения получают от пользователей маркеры доступа, например, в формате *JSON Web Tokens*, которые проверяются через поставщика удостоверений.

Рассмотрение протоколов делегирования децентрализованного доступа и их реализации выходит за рамки этой книги. Задача корпоративных систем заключается в перехвате и перенаправлении аутентификации пользователя поставщику удостоверений. В зависимости от системной архитектуры эта ответственность возлагается на прокси-сервер или на само приложение.

Существуют решения с открытым исходным кодом, которые обеспечивают децентрализованную безопасность. Интересной технологией является *Keycloak*,

решение Identity and Access Management. Оно поставляется с различными клиентскими адаптерами и поддерживает стандартные протоколы, такие как OAuth и OpenID Connect, что упрощает защиту приложений и сервисов.

Прокси-серверы

Прокси-серверы, инкапсулирующие связь с корпоративными приложениями, могут дополнительно реализовывать такие аспекты безопасности, как шифрование связи. Например, веб-прокси-серверы поддерживают TLS-шифрование через протокол HTTPS.

Вопрос в том, хотят ли инженеры, чтобы существовали различия между внутренней и внешней сетевой коммуникацией. Связь по внутренней сети часто не шифруется. В зависимости от характера обмена информацией связь по Интернету в большинстве случаев должна быть зашифрована.

Прокси-серверы могут использоваться для отмены шифрования в границах сети — так называемого *TLS-аннулирования*. Прокси-сервер шифрует всю исходящую и дешифрует всю входящую информацию.

Также можно повторно шифровать соединение, применяя разные сертификаты для разных сетей.

Интеграция в современных средах

Современные среды нацелены на поддержку существующих на данный момент потребностей в области безопасности. Фреймворки управления контейнерами предоставляют программные прокси-серверы и шлюзы с таким сервисом. Например, ресурс Kubernetes Ingress, а также OpenShift Routes поддерживают TLS-шифрование для внешнего трафика кластера.

Для того чтобы предоставлять секретные значения, например регистрационные данные или закрытые ключи, фреймворки управления реализуют свои функции обеспечения безопасности. Как было показано ранее, это позволяет поставлять в среду секретные конфигурации отдельно от остальных данных. Подробнее данный процесс описан в главе 5. Это позволяет приложениям, а также всей конфигурации использовать секретные значения. При необходимости секретные данные могут быть внедрены во время работы контейнера.

Обеспечение безопасности в приложениях Java EE

Ранее речь шла о наиболее распространенных современных подходах к обеспечению безопасности, теперь посмотрим, как она реализована в Java EE.

Из всех версий Java версия 8 Java EE больше всего ориентирована на решение проблем безопасности. Она содержит API безопасности, который упрощает и унифицирует интеграцию для программистов.

Прозрачная безопасность

Проще всего безопасность в веб-приложениях можно реализовать через прокси-серверы, такие как *Apache* или *nginx*. В этом случае задачи безопасности являются прозрачными для приложения. Так часто случается, когда корпоративное приложение не должно иметь дела с пользователями как объектами предметной области.

Сервлеты

Для того чтобы защитить веб-сервисы, реализованные в приложении Java EE, обычно используют безопасность на уровне сервлета. Это относится ко всем технологиям, построенным на основе сервлетов, таким как JAX-RS. Конфигурация функций безопасности описывается с применением дескриптора развертывания сервлета, то есть в файле `web.xml`. Это можно сделать несколькими способами, такими как аутентификация на основе форм, на базе доступа HTTP или клиентских сертификатов.

Аналогично решения безопасности, такие как Keycloak, поставляются с собственными адаптерами и фильтрами сервлетов. Программистам обычно остается лишь настроить конфигурацию этих компонентов для использования поставщика услуг по безопасности.

Субъекты и роли Java

Субъекты и роли безопасности Java представляют собой сущности и роли авторизации соответственно. Конфигурации субъектов и ролей обычно описываются на сервере приложений, способ описания зависит от поставщика. Аутентифицированные запросы привязываются к субъекту во время выполнения.

Одним из примеров использования связанных ролей в процессе выполнения является применение общих аннотаций безопасности, таких как `@RolesAllowed`. Этот декларативный подход проверяет правильность авторизации субъекта и, если проверка не пройдена, выдает исключение безопасности:

```
import javax.annotation.security.RolesAllowed;

@Stateless
public class CarManufacturer {

    …

    @RolesAllowed("worker")
    public Car manufactureCar(Specification spec) {
        …
    }

    @RolesAllowed("factory-admin")
    public void reconfigureMachine(...) {
        …
    }
```

Пользователи и роли могут быть расширены и, помимо решений для конкретных поставщиков, станут включать в себя информацию о предметной области. Для этого в примере был создан расширенный тип безопасности `Principal`.

Можно внедрить этот субъект, идентифицируемый по имени, и предоставить его специализацию. Контейнер выполняет идентификацию пользователя, например реализуя проверку подлинности на основе формы.

Этот подход настоятельно рекомендуется использовать в приложениях до версии Java EE 8. Однако в современных приложениях для представления пользовательской информации о конкретной предметной области, скорее всего, будут применяться хранилища идентификаторов.

JASPIC

Интерфейс поставщика услуг аутентификации Java для контейнеров *(Java Authentication Service Provider Interface for Containers, JASPIC)* — это стандарт, определяющий интерфейсы поставщиков услуг аутентификации. Он включает в себя так называемые модули аутентификации сервера *(Server Authentication Modules, SAM)* — подключаемые компоненты аутентификации, которые устанавливаются на сервер приложений.

Этот стандарт предлагает мощные и гибкие способы реализации аутентификации. Поставщики серверов могут использовать собственную реализацию модулей SAM. Однако внедрение модулей аутентификации с применением стандарта JASPIC многие программисты считают довольно громоздким. Поэтому стандарт JASPIC не получил широкого распространения в корпоративных проектах.

Security API

Security API 1.0 поставляется в комплекте с Java EE 8. Смысл этого стандарта заключается в предоставлении современных методов обеспечения безопасности, которые программистам будет проще использовать. Они реализованы независимо от поставщиков, причем не требуют блокировки определенных решений.

Рассмотрим, что включает в себя Security API.

Механизмы аутентификации

Прежде всего, Security API включает в себя `HttpAuthenticationMechanism`, обеспечивающий функции стандарта JASPIC с гораздо меньшими затратами на разработку. Он рассчитан на применение в контексте сервлета.

Разработчикам приложений требуется только определить пользовательский `HttpAuthenticationModule` и описать конфигурацию аутентификации в дескрипторе развертывания `web.xml`. В этой главе мы рассмотрим нестандартную реализацию обеспечения безопасности.

Контейнер Java EE поставляется в комплекте с готовыми механизмами аутентификации HTTP для базовой, стандартной и специальной аутентификации на основе формы. Программисты могут использовать эту предопределенную функциональность с минимальными изменениями. Прежде чем рассмотреть пример, изучим, как хранится информация о пользователе.

Хранилище удостоверений

Вместе с Security API появилась концепция хранилищ удостоверений. Хранилища удостоверений предоставляют информацию об аутентификации и авторизации пользователей в легком, переносимом виде. Они обеспечивают единый способ доступа к ней.

Тип `IdentityStore` проверяет регистрационные данные вызывающего абонента и получает доступ к его информации. Подобно механизмам аутентификации HTTP, контейнеры приложений должны предоставлять хранилища идентификаторов для доступа к LDAP и базе данных.

Далее приведен пример использования предоставляемых контейнером функций обеспечения безопасности:

```
import javax.security.enterprise.authentication.mechanism.http.*;
import javax.security.enterprise.identitystore.DatabaseIdentityStoreDefinition;
import javax.security.enterprise.identitystore.IdentityStore;

@BasicAuthenticationMechanismDefinition(realmName = "car-realm")
@DatabaseIdentityStoreDefinition(
        dataSourceLookup = "java:comp/UserDS",
        callerQuery = "select password from users where name = ?",
        useFor = IdentityStore.ValidationType.VALIDATE
)
public class SecurityConfig {
    // не менять конфигурацию
}
```

Разработчики приложений должны только предоставить этот аннотированный класс. Метод обеспечивает простые и понятные определения безопасности для тестирования.

Обычные корпоративные проекты иногда требуют более индивидуального подхода. У каждой организации, как правило, есть свои способы аутентификации и авторизации, которые необходимо интегрировать.

Специальные меры безопасности

Далее приведен более сложный пример.

Для того чтобы обеспечить нестандартную аутентификацию, разработчики приложений реализуют специальный класс `HttpAuthenticationMechanism`, а в нем — метод `validateRequest()`. Класс должен быть видимым только в контейнере — как CDI-компонент. Остальное выполняет контейнер приложения. Это упрощает программистам работу по интеграции средств безопасности.

Вот простейший пример реализации аутентификации на *псевдокоде*:

```
import javax.security.enterprise.AuthenticationException;
import javax.security.enterprise.authentication.mechanism.http.*;
import javax.security.enterprise.credential.UsernamePasswordCredential;
import javax.security.enterprise.identitystore.CredentialValidationResult;
import javax.security.enterprise.identitystore.IdentityStoreHandler;

@ApplicationScoped
public class TestAuthenticationMechanism implements
        HttpAuthenticationMechanism {

    @Inject
    IdentityStoreHandler identityStoreHandler;

    @Override
    public AuthenticationStatus validateRequest(HttpServletRequest request,
            HttpServletResponse response,
            HttpMessageContext httpMessageContext)
            throws AuthenticationException {

        // получить данные аутентификации
        String name = request.get…
        String password = request.get…

        if (name != null && password != null) {

            CredentialValidationResult result = identityStoreHandler
                .validate(new UsernamePasswordCredential(name,
                password));

            return httpMessageContext.notifyContainerAboutLogin(result);
        }
        return httpMessageContext.doNothing();
    }
}
```

Метод `validateRequest()` получает доступ к информации пользователя, содержащейся в HTTP-запросе, например, через HTTP-заголовки. Он делегирует проверку хранилищу удостоверений с помощью `IdentityStoreHandler`. Результат проверки содержится в теле HTTP-сообщения безопасности.

В зависимости от требований может также понадобиться специальная реализация обработчика идентификаторов, обеспечивающая нестандартные методы аутентификации и авторизации.

Если используются децентрализованные протоколы безопасности, такие как OAuth, специальный обработчик идентификаторов будет проверять маркер доступа к данным безопасности.

Далее показана реализация специального хранилища удостоверений:

```
import javax.security.enterprise.identitystore.IdentityStore;

@ApplicationScoped
public class TestIdentityStore implements IdentityStore {

    public CredentialValidationResult validate(UsernamePasswordCredential
```

```
            usernamePasswordCredential) {

        // специальная аутентификация и авторизация
        // если проверка пройдена

        return new CredentialValidationResult(username, roles);

        // если регистрационные данные неверны

        return CredentialValidationResult.INVALID_RESULT;
    }
}
```

Дескриптор развертывания сервлета `web.xml` используется для указания защищенных ресурсов. Задача интеграции возлагается на контейнер приложений:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Protected pages</web-resource-name>
        <url-pattern>/management</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin-role</role-name>
    </auth-constraint>
</security-constraint>
```

Механизм аутентификации HTTP обеспечивает простой, но гибкий способ реализации безопасности JASPIC. Эта реализация проще, чем непосредственный JASPIC. Она обеспечивает возможность перехвата коммуникационных потоков и может интегрировать приложение со сторонними продуктами обеспечения безопасности.

Доступ к секретной информации

Корпоративным приложениям иногда требуется функциональность для доступа к информации об авторизации пользователя в рамках бизнес-логики. Security API предоставляет единообразный способ получения этой информации.

Он содержит тип `SecurityContext`, обеспечивающий программный способ получения информации о главном вызывающем абоненте и его ролях. `SecurityContext` внедряется в любые управляемые компоненты. Он также интегрируется с конфигурацией аутентификации сервлетов и сообщает о том, разрешен ли вызывающему абоненту доступ к определенному HTTP-ресурсу.

Далее приведен пример использования `SecurityContext`:

```
import javax.security.enterprise.SecurityContext;

@Stateless
public class CompanyProcesses {

    @Inject
    SecurityContext securityContext;

    public void executeProcess() {
        executeUserProcess();
```

```
        if (securityContext.isCallerInRole("admin")) {
            String name = securityContext.getCallerPrincipal().getName();
            executeAdminProcess(name);
        }
    }

    …
}
```

Идея Security API заключается в интеграции с существующей функциональностью из предыдущих версий Java EE. Это означает, например, что аннотация `@RolesAllowed` использует ту же информацию о роли, что и `SecurityContext`. Программисты могут полагаться на существующую стандартную функциональность.

Резюме

В современном мире информационная безопасность очень важна. В прошлом самыми большими проблемами безопасности были слабость алгоритмов шифрования и хеширования, сохранение паролей и самодельные реализации систем безопасности. Основными принципами безопасности являются шифрование связи с применением надежных внешних поставщиков средств обеспечения безопасности для аутентификации и авторизации без сохранения регистрационных данных в системе контроля версий и реализация тестовых сценариев, проверяющих защиту.

Связь обычно шифруется на транспортном уровне с использованием TLS. Сертификаты должны быть правильно подписаны либо корпоративным, либо официальным органом сертификации. Другие методы включают в себя реализацию функций безопасности на уровне протокола, таких как базовая аутентификация HTTP на основе шифрованной связи.

Децентрализованная безопасность отделяет функции аутентификации и авторизации от приложений, используя доверенных поставщиков удостоверений. Примером этого могут служить системы единого входа и децентрализованные протоколы делегирования доступа.

Безопасность в границах приложений Java EE обычно реализуется на базе сервлетов. Security API, задействованный в Java EE 8, призван обеспечить более простые и единообразные методы решения задач безопасности в приложениях Java EE. Механизмы аутентификации HTTP — это пример более простого применения мощных функций JASPIC. Хранилища удостоверений предоставляют информацию об аутентификации и авторизации пользователей.

Суть Security API заключается в интеграции с существующей функциональностью и обеспечении единообразных механизмов доступа. Его функций должно быть достаточно для защиты корпоративного приложения на стороне HTTP.

11 Заключение

Я надеюсь, читая эту книгу, вы получили полезные знания о том, как создавать современные, легкие, бизнес-ориентированные корпоративные приложения. Возможно, она даже позволит вам отказаться от парочки устаревших методов разработки, которые были хороши в прошлом.

Мы увидели, как современные версии Java EE вписываются в новый мир производства программного обеспечения с контейнерными технологиями, облачными платформами, автоматизацией, непрерывной поставкой кода и многим другим.

Правильная постановка задач при корпоративной разработке

Как мы уже неоднократно видели, команды инженеров должны быть правильно мотивированы при разработке программного обеспечения. Если это корпоративные системы, то основное внимание следует уделять бизнес-мотивации. Сценарии предметной области и бизнес-сценарии должны быть понятными, чтобы приносить пользу клиентам. В конце концов, именно программное обеспечение, выполняющее свои бизнес-функции, является источником дохода.

Со временем разработчикам полезно задать себе вопрос: *помогает ли то, что мы делаем, решить задачу бизнеса*?

Программное обеспечение, предназначенное для удовлетворения потребностей клиента, фокусируется главным образом на выполнении бизнес-сценариев. Технология, реализующая сопутствующие потребности, такие как связь, сохранение или распространение данных, играет второстепенную роль. Выбранные решения должны быть направлены прежде всего на реализацию бизнес-логики.

Поэтому технологии, языки программирования и фреймворки в идеале должны поддерживать реализацию бизнес-сценариев без чрезмерных затрат. Группе инженеров рекомендуется выбирать технологию, с которой они знакомы и могут продуктивно работать и которая также соответствует этому требованию.

Облачные среды и непрерывная поставка

Мы увидели необходимость быстрых изменений в быстро меняющемся мире. Важно акцентировать внимание на гибкости и быстрой реакции на потребности клиента, времени выхода на рынок или, еще лучше, *времени производства*. Самые совершенные функции не принесут дохода, пока не попадут в руки клиента.

Имеет смысл использовать концепции и технологии, которые помогают достичь этой цели: непрерывную поставку, автоматизацию, инфраструктуру как код и автоматизированные программные тесты. Именно способность быстро меняться является главным преимуществом современных сред и облачных технологий. Среды приложений для новых проектов, функций и тестовых сценариев могут быть созданы за считаные минуты по четко определенным спецификациям. В частности, эту концепцию поддерживают инфраструктура как код и контейнерные технологии. Разработчики программного обеспечения предоставляют конфигурацию среды вместе с кодом приложения, которые содержатся в репозитории проекта.

Таким образом, за определение всего содержимого корпоративного программного обеспечения отвечает вся группа инженеров. И программисты, и инженеры по эксплуатации заинтересованы в поставке программного обеспечения, которое представляет ценность для пользователей. За достижение этой цели отвечает вся команда разработчиков.

Сюда также входят задачи обеспечения качества программного обеспечения. Быстрая доставка функций возможна только при наличии правильных автоматических механизмов проверки качества. Тесты, которые требуют вмешательства человека и работают ненадежно или недостаточно быстро, замедляют процессы и не позволяют разработчикам выполнять более полезную работу. Необходимо стремиться к созданию автоматизированных, достаточных и надежных тестовых примеров, построенных с учетом удобства обслуживания и обеспечения качества кода.

Актуальность Java EE

Мы увидели, что Java EE позволяет все это делать. Платформа поддерживает фокусирование на бизнес-требованиях, позволяя программистам писать код, и не устанавливает слишком много ограничений. Можно проектировать и реализовывать бизнес-сценарии, опираясь в первую очередь на требования предметной области.

Сама технология *не требует внимания*. В большинстве случаев достаточно аннотировать бизнес-логику, и контейнер приложения сам добавит все, что необходимо с технической точки зрения. Концепции стандартов Java EE, такие как

JAX-RS, JPA и JSON-B, обеспечивают необходимую техническую интеграцию при минимуме усилий.

Особенно легко платформа Java EE позволяет инженерам интегрировать несколько стандартов без дополнительной настройки. Это стало возможным благодаря спецификациям JSR, написанным с учетом принципов Java EE.

Современная Java EE радикально отличается от прежней J2EE. Фактически ее модель программирования и среда выполнения имеют мало общего с J2EE.

Благодаря обратной совместимости платформы устаревшие методы все еще действуют, но с тех пор технология ушла далеко вперед. Были пересмотрены и значительно упрощены модели программирования и шаблоны проектирования. В частности, исчезли прежние ограничения шаблонов при реализации иерархий технологически обоснованных интерфейсов, а также суперклассы. Программисты теперь могут сосредоточиться на бизнес-логике, а не на технологии.

Характер стандартов Java EE позволяет компаниям реализовывать приложения, независимые от поставщиков, и, соответственно, избежать технологической зависимости от них. Программистам также не приходится специально обучаться технологиям определенных поставщиков. Нам встречалось немало случаев, когда команды были знакомы только с определенными продуктами, которые со временем устарели.

Технология Java EE используется не только на стороне сервера. Такие стандарты, как JAX-RS, JSON-P и CDI, являются важными преимуществами и для приложений Java SE. Имеет смысл реализовать определенные функции, такие как HTTP-клиенты, на базе стандартных технологий, знакомых программистам.

Обновления API в Java EE 8

В книге основное внимание уделено корпоративным приложениям на базе Java EE 8.

В этой версии был обновлен ряд стандартов. Далее перечислены наиболее важные из них, а также новые функции.

CDI 2.0

Начиная с Java EE 8 и CDI 2.0, события могут обрабатываться не только синхронно. Как мы видели, CDI изначально поддерживает асинхронную обработку событий. В действительности это было возможно только до того, как метод-наблюдатель событий стал бизнес-методом EJB с аннотацией `@Asynchronous`.

Для того чтобы создавать и обрабатывать асинхронные события CDI, на стороне издателя используют метод `fireAsync`. Параметр метода-наблюдателя снабжается аннотацией `@ObservesAsync`.

Еще одна новая функциональность для обработки событий, появившаяся в CDI 2.0, — это возможность назначать наблюдателей событий. Для того чтобы

назначить метод — наблюдатель за событиями, применяется аннотация `@Priority`, хорошо известная на платформе Java EE:

```
public void onCarCreated(@Observes @Priority(100) CarCreated event) {
    System.out.println("first: " + newCoffee);
}

public void alsoOnCarCreated(@Observes @Priority(200) CarCreated event)
{
    System.out.println("second: " + newCoffee);
}
```

Такой подход гарантирует, что наблюдатели событий будут вызваны в указанном порядке, начиная с более низкого приоритета. Программисты должны проверить, нарушает ли эта ситуация принципы позднего связывания и единой точки ответственности путем изменения последовательности вызова обработчиков событий.

Главной особенностью CDI 2.0 стала интеграция за пределами корпоративного контейнера, обеспечивающая возможность задействования CDI в приложениях Java SE. Смысл заключается в том, что приложения Java SE могут использовать также функции сложного стандарта внедрения зависимостей, что позволяет расширить применение CDI за пределы мира Java EE.

JAX-RS 2.1

Версия JAX-RS 2.1 была рассчитана главным образом на реактивных клиентов SSE и предназначалась для лучшей интеграции с такими стандартами, как JSON-B. Помимо этого, в нее были добавлены еще некоторые небольшие улучшения.

Реактивное программирование используется все шире. В частности, клиент получает новые реактивные функции для выполнения HTTP-вызовов и прямого возврата так называемых реактивных типов. Примером такого типа является `CompletionStage`. Этот тип поддерживается изначально, другие типы и библиотеки могут быть добавлены через расширения.

Для совершения реактивных вызовов применяется метод `rx()` для `Invocation.Builder`.

Как показано в этой книге, JAX-RS 2.1 поддерживает SSE и на стороне клиента, и на стороне сервера. Стандарт SSE представляет собой легкий односторонний протокол обмена сообщениями, в котором задействуются сообщения в формате открытого текста по протоколу HTTP.

Для того чтобы соответствовать основным концепциям платформы Java EE, стандарт JSON-B, добавленный в Java EE 8, легко интегрируется в JAX-RS. Это означает, что, подобно JAXB, типы Java, которые используются в качестве тел запроса или ответа, неявно преобразуются в формат JSON.

Аналогично новые функции, которые являются частью JSON-P 1.1 и Bean Validation 2.0, включены в JAX-RS. Это возможно, поскольку спецификации передают определенные функции в соответствующие стандарты.

Еще одно небольшое обновление, вошедшее в состав в JAX-RS, включало в себя аннотацию `@PATCH` для одноименного метода HTTP. Несмотря на то что в JAX-RS и раньше была возможна поддержка других HTTP-методов, это упрощает их использование для программистов, которым требуется данная функция.

Еще одним небольшим, но действительно полезным улучшением было включение стандартизированных методов задержки HTTP в клиент JAX-RS. Методы компоновщика `connectTimeout` и `readTimeout` поддерживают настраиваемые задержки. Такая конфигурация требуется во многих проектах, прежде для этого требовалось подключение сторонних функций.

Реализация этих функций была описана в главе 3.

JSON-B 1.0

JSON-B — это новый стандарт, который преобразует типы Java в структуры JSON и обратно. Подобно JAXB для XML, он предоставляет функциональность для декларативного преобразования объектов.

Самым большим преимуществом этого стандарта в экосистеме Java EE является то, что приложения больше не должны полагаться на сторонние реализации. Фреймворки преобразования JSON обычно препятствуют сборке переносимых корпоративных приложений. Они повышают риск нарушения зависимостей от существующих версий фреймворка в среде выполнения.

JSON-B решает эту проблему, обеспечивая стандартное преобразование JSON. Распространение нестандартных фреймворков преобразования, таких как Jackson или Johnzon, больше не требуется.

JSON-P 1.1

Стандарт JSON-P 1.0, представленный в Java EE 7, обеспечивает мощную возможность программно создавать и читать структуры JSON. Версия 1.1 в основном включает в себя поддержку общих стандартов JSON.

Одним из этих стандартов IETF является *JSON Pointer* (RFC 6901). Он определяет синтаксис для запроса структур и значений JSON. Используя указатели, такие как `"/0/user/address"`, можно ссылаться на значения JSON, как на *XPath* в XML.

Эта функция включена в тип `JsonPointer`, создаваемый с помощью метода `Json.createPointer()`, аналогично существующему JSON-P API.

Другим с недавних пор поддерживаемым стандартом является *JSON Patch* (RFC 6902). Он определяет так называемые патчи и методы модификации, которые применяют к существующим структурам JSON.

JSON 1.1 поддерживает создание патчей JSON с помощью `Json.createPatch` или `Json.createPatchBuilder`. Соответствующий тип JSON-P называется `JsonPatch`.

Третий стандарт, поддерживаемый IETF, — это *JSON Merge Patch* (RFC 7386). Он позволяет объединять существующие структуры JSON для создания новых структур. JSON-P поддерживает создание патчей слияния посредством

`Json.createMergeDiff` или `Json.createMergePatch` и получение в результате типа `JsonMergePatch`.

Помимо этих стандартов, поддерживаемых IETF, JSON-P 1.1 включает в себя несколько небольших функций, упрощающих использование API. Одним из примеров такой функции является поддержка потоков Java SE 8 через предопределенные коллекторы потоков, такие как метод `JsonCollectors.toJsonArray()`. Еще одно небольшое улучшение позволяет создавать типы значений JSON-P из строк и примитивов Java посредством `Json.createValue`.

Bean Validation 2.0

В Java EE 8 версия Bean Validation обновилась до 2.0. Помимо новых предопределенных ограничений главным изменением в ней стала поддержка Java SE 8.

Поддержка Java SE 8 включает в себя несколько аннотаций с разной конфигурацией ограничений проверки. Теперь поддерживаются типы API Java 8 Date и Time — например, их можно использовать как `@Past LocalDate date`.

Значения, которые содержатся в контейнерных типах, также могут быть проверены по отдельности через аннотации параметризованных типов, например `Map<String, @Valid Customer> customers`, `List<@NotNull String> strings` и `Optional<@NotNull String> getResult()`.

В Bean Validation 2.0 появились новые предопределенные ограничения. Например, `@Email` проверяет адреса электронной почты, а `@Negative` и `@Positive` — числовые значения, `@NotEmpty` гарантирует, что коллекции, карты, массивы и строки не являются пустыми и их значение не равно `null`, `@NotBlank` проверяет, чтобы строка не состояла только из пробелов.

Эти ограничения являются полезной стандартной функцией, которая позволяет избежать выполнения подобных проверок вручную.

JPA 2.2

В Java EE 8 спецификация JPA обновилась до версии 2.2. Эта версия ориентирована в основном на функции Java SE 8.

Подобно Bean Validation, поддержка Java SE 8 включает в себя API Date и Time. Такие типы, как `LocalDate` или `LocalDateTime`, теперь поддерживаются по умолчанию для свойств сущностей.

Версия 2.2 позволяет возвращать результат запроса не только в формате `List<T>`, но и как `Stream<T>`, используя метод `getResultStream()`, как показано в следующем фрагменте кода:

```
Stream<Car> cars = entityManager
        .createNamedQuery(Car.FIND_TWO_SEATERS, Car.class)
        .getResultStream();
cars.map(...)
```

Кроме того, в JPA 2.2 наконец появилась поддержка внедрения управляемых компонентов в преобразователи атрибутов с помощью CDI `@Inject`. Это увеличи-

вает количество сценариев нестандартных преобразователей атрибутов и делает их более полезными. Подобно другим стандартам, таким как JSON-B, улучшенная интеграция CDI поощряет повторное применение компонентов Java EE.

Также в версии 2.2 появились многократно используемые аннотации, такие как `@JoinColumn`, `@NamedQuery` и `@NamedEntityGraph`. Поскольку Java SE 8 позволяет применять один и тот же тип аннотации несколько раз, программистам больше не приходится задействовать для этого соответствующие групповые аннотации, такие как `@JoinColumns`.

Security 1.0

Как видно из последней главы, целью Security 1.0 является упрощение интеграции задач безопасности в приложения Java EE. Поэтому программистам предлагается задействовать мощные функции, такие как JASPIC.

В главе 10 мы рассмотрели функции и применение механизмов аутентификации HTTP, хранилищ удостоверений и контексты безопасности.

Servlet 4.0

На момент написания этой книги HTTP/1.1 был главной используемой версией HTTP. Основной целью HTTP/2 является устранение недостаточной производительности HTTP в веб-приложениях. В частности, запрос нескольких ресурсов веб-системы мог привести к неоптимальной производительности из-за многочисленных подключений. Версия HTTP 2 направлена на уменьшение простоев и повышение пропускной способности путем мультиплексирования, конвейерной обработки, сжатия заголовков и отправки данных по инициативе сервера (server push).

Большинство изменений в HTTP/2 не влияет на работу инженеров — совсем не так, как было в версии 1.1. Контейнер сервлета сам обрабатывает все, что касается HTTP. Исключением является функция `Server Push`.

`Server Push` работает таким образом, что сервер напрямую отправляет HTTP-ответы ресурсам, связанным с тем ресурсом, который запросил клиент, исходя из предположения, что клиенту понадобятся и они тоже. Это позволяет серверу отправлять ресурсы, которые не были запрошены клиентом явно. На веб-страницах этот метод оптимизации производительности в основном применяется к таблицам стилей, коду JavaScript и другим активам.

Servlet API поддерживает сообщения `Server Push` с помощью типа `PushBuilder`, создаваемого методом `HttpServletRequest.newPushBuilder()`.

JSF 2.3

Java SErver Faces — это традиционный способ построения сервер-ориентированных пользовательских интерфейсов на основе компонентов HTML. Java EE 8 поставляется в комплекте с обновленной версией JSF 2.3.

Основные усовершенствования в этой версии включают в себя лучшую интеграцию CDI, WebSocket и AJAX, Bean Validation на уровне классов, а также поддержку Java SE 8.

Поскольку основное внимание в этой книге уделяется серверной части, то о JSF здесь упоминается только вскользь.

JCP и участие в создании стандартов

Java Community Process (JCP) определяет стандарты, которые образуют платформы Java SE и EE, включая сам обобщающий стандарт Java EE. Отдельные стандарты определяются как *Java Specification Requests (JSR)*, каждый из которых образует так называемые *экспертные группы*, состоящие из экспертов и компаний, занимающихся разработкой корпоративного программного обеспечения.

Идея заключается в стандартизации технологии, которая хорошо себя зарекомендовала в реальных проектах. Объединяя опыт компаний и отдельных лиц из этих реальных проектов, удается сформировать независимые от вендора корпоративные стандарты Java.

Очень важно, чтобы в JCP участвовали как компании, так и частные лица. Это дает возможность формировать стандарты и определять будущее технологии Java, а также приобретать знания в области этой технологии. Открытые процессы JCP позволяют программистам узнать, какими будут следующие версии Java EE.

Частные лица и компании также могут следить за процессами стандартизации, даже если не участвуют в JCP. Они могут просматривать рабочие версии стандартов и отправлять свои соображения экспертным группам.

Экспертные группы активно приветствуют конструктивную обратную связь при разработке спецификаций. Очень полезно получать отзывы, сведения об опыте применения технологии в реальных проектах и помощь в разработке стандартов, лучше отвечающих потребностям отрасли.

Я тоже участвовал в составлении Java EE 8 — входил в две экспертные группы: JAX-RS 2.1 и JSON-P 1.1. И многому научился в ходе этой работы. Приглашаю других разработчиков корпоративных Java-проектов участвовать в процессах JCP.

MicroProfile

Целью инициативы MicroProfile было создать надстройку над стандартами Java EE и построить профили меньшего масштаба, предназначенные для микросервисных архитектур, а также поэкспериментировать с функциями, не зависящими от стандартизации. В этой инициативе принимали участие несколько поставщиков серверов приложений, которые формируют фактические стандарты, согласованные с прочими поставщиками.

Главное преимущество серверных приложений, поддерживающих MicroProfile, — возможность запуска приложений Java EE, требующих лишь небольшого количества стандартов (в первой версии это JAX-RS, CDI и JSON-P). Подобным образом поставщики серверов приложений позволяют сократить среду выполнения до определенного набора стандартов.

Преимущество этих подходов заключается в том, что они не просто оптимизируют время выполнения, но и не добавляют для этого новые зависимости в корпоративный проект. Программисты могут продолжать писать приложения, используя стандартную технологию Java EE.

Eclipse Enterprise for Java

В сентябре 2017 года, перед самой публикацией этой книги, компания Oracle, владелец технологий Java EE и JCP, объявила о передаче платформы Java EE и ее стандартов организации Open Source Foundation, что стало причиной появления *Eclipse Enterprise for Java (EE4J)*. Сейчас планируется снизить барьер для компаний и программистов, которые хотели бы внести свой вклад в создание более открытой технологии.

Однако, как бы ни выглядела реализация этих планов, важно отметить, что они подразумевают сохранение сути платформы. Концепции и методы, представленные в этой книге, сохранятся в будущих версиях корпоративной Java.

Я могу повторить то, что уже говорил об участии в JCP. Но, поскольку запущен процесс стандартизации Enterprise Java, я призываю инженеров и компании присмотреться к Eclipse Enterprise for Java и принять участие в разработке корпоративных стандартов. Коллективные знания и практический опыт помогли сформировать стандарты Java EE, помогут создать их и для Enterprise Java.

Приложение. Дополнительные ресурсы

В этой книге мы рассмотрели множество тем, относящихся к Java EE. В процессе написания я пользовался литературой из нескольких источников. Продолжить изучение этого предмета вам помогут следующие ресурсы (указаны в том порядке, в каком были использованы в книге).

- Java Enterprise Platform: http://www.oracle.com/technetwork/java/javaee/overview/index.html.
- Java Community Process: https://jcp.org/en/home/index.
- Robert C. Martin (Uncle Bob). Clean Code.
- Erich Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software.
- Eric Evans. Domain-Driven Design.
- Robert C. Martin (Uncle Bob). Screaming Architecture: https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html.
- Mel Conway. Conway's Law: http://www.melconway.com/Home/Conways_Law.html.
- Apache Maven: https://maven.apache.org.
- Gradle: https://gradle.org.
- Servlet API 4: https://www.jcp.org/en/jsr/detail?id=369.
- Ivar Jacobson. Entity Control Boundary.
- Java EE 8 (JSR 366): https://jcp.org/en/jsr/detail?id=366.
- Enterprise JavaBeans 3.2 (JSR 345): https://jcp.org/en/jsr/detail?id=345.
- Context and Dependency Injection for Java 2.0 (JSR 365): https://jcp.org/en/jsr/detail?id=365.
- Simple Object Access Protocol (SOAP): https://www.w3.org/TR/soap/.

- Java API for RESTful Web Services 2.1 (JSR 370): https://jcp.org/en/jsr/detail?id=370.
- Roy T. Fielding. Architectural Styles and the Design of Network-based Software Siren: https://github.com/kevinswiber/siren.
- Java API for JSON Binding 1.0 (JSR 367): https://jcp.org/en/jsr/detail?id=367.
- Java API for JSON Processing 1.1 (JSR 374): https://jcp.org/en/jsr/detail?id=374.
- Java XML Binding 2.0 (JSR 222): https://jcp.org/en/jsr/detail?id=222.
- Bean Validation 2.0, (JSR 380): https://jcp.org/en/jsr/detail?id=380.
- Java Message Service 2.0 (JSR 343): https://jcp.org/en/jsr/detail?id=343.
- Server-Sent Events: https://www.w3.org/TR/eventsource/.
- WebSocket Protocol (RFC 6455): https://tools.ietf.org/html/rfc6455.
- Java API for WebSocket (JSR 356): https://jcp.org/en/jsr/detail?id=365.
- Enterprise JavaBeans/Interceptors API 1.2 (JSR 318): https://jcp.org/en/jsr/detail?id=318.
- Java Temporary Caching API (JSR 107): https://jcp.org/en/jsr/detail?id=107.
- MicroProfile: https://microprofile.io.
- Docker Documentation: https://docs.docker.com.
- Kubernetes Documentation: https://kubernetes.io/docs/home.
- OpenShift Documentation: https://docs.openshift.com.
- Cloud Native Computing Foundation: https://www.cncf.io.
- 12-факторные приложения: https://12factor.net.
- Kevin Hoffman. Beyond the 12 Factor App: https://content.pivotal.io/ebooks/beyond-the-12-factor-app.
- Jenkins: https://jenkins.io.
- Using a Jenkinsfile, Documentation: https://jenkins.io/doc/book/pipeline/jenkinsfile.
- Semantic Versioning: http://semver.org.
- JUnit 4: http://junit.org/junit4.
- Mockito: http://site.mockito.org.
- Arquillian: http://arquillian.org.
- CDI-Unit: https://bryncooke.github.io/cdi-unit.
- AssertJ: http://joel-costigliola.github.io/assertj.
- TestNG: http://testng.org/doc.
- WireMock: http://wiremock.org.
- Gatling: https://gatling.io.
- Apache JMeter: http://jmeter.apache.org.
- Cucumber-JVM: https://cucumber.io/docs/reference/jvm.
- FitNesse: http://fitnesse.org.

- Prometheus: https://prometheus.io.
- Grafana: https://grafana.com.
- fluentd: https://www.fluentd.org.
- Chronicle Queue: http://chronicle.software/products/chronicle-queue.
- OpenTracing: http://opentracing.io.
- AsciiDoc: http://asciidoc.org.
- Markdown: https://daringfireball.net/projects/markdown.
- OpenAPI: https://www.openapis.org.
- Swagger: https://swagger.io.
- Adam Bien. Porcupine: https://github.com/AdamBien/porcupine.
- Adam Bien. Breakr: https://github.com/AdamBien/breakr.
- OWASP: https://www.owasp.org.
- OAuth: https://oauth.net.
- OpenID: https://openid.net.
- JSON Web Tokens: https://jwt.io.
- Java Authentication Service Provider Interface for Containers (JSR 196): https://www.jcp.org/en/jsr/detail?id=196.