Предметно-ориентированное проектирование в Enterprise Java

Предметно-ориентированное проектирование в Enterprise Java

с помощью Jakarta EE, Eclipse MicroProfile, Spring Boot и программной среды Axon Framework

Виджей Наир

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Виджей Наир

Предметно-ориентированное проектирование в Enterprise Java с помощью Jakarta EE, Eclipse MicroProfile, Spring Boot и программной среды Axon Framework

Practical Domain-Driven Design in Enterprise Java

Using Jakarta EE, Eclipse MicroProfile, Spring Boot, and the Axon Framework

Vijay Nair

Apress®

Предметно-ориентированное проектирование в Enterprise Java с помощью Jakarta EE, Eclipse MicroProfile, Spring Boot и программной среды Axon Framework

Виджей Наир

ДМК Пресс
издательство
Москва, 2020

УДК 004.42Java
ББК 32.972
Н20

Наир В.
Н20 Предметно-ориентированное проектирование в Enterprise Java с помощью Jakarta EE, Eclipse MicroProfile, Spring Boot и программной среды Axon Framework / пер. с англ. А. В. Снастина. – М.: ДМК Пресс, 2020. – 306 с.: ил.

ISBN 978-5-97060-872-2

В книге подробно рассматриваются реализации шаблонов предметно-ориентированного проектирования с применением различных инструментальных средств и рабочих программных сред из Enterprise Java Space. При таком подходе читатель получает завершенную картину и возможность практического применения любого из этих средств в процессе предметно-ориентированного проектирования.

В начальных главах описывается эталонная реализация проекта Cargo Tracker – создание монолитного приложения с использованием платформы Jakarta EE. Затем рассматривается полный процесс преобразования монолитного приложения в архитектуру, основанную на микросервисах. В заключительных главах демонстрируется создание версии приложения с использованием шаблонов CQRS и Event Sourcing (ES); основной рабочей средой является Axon Framework.

Издание будет полезно специалистам, приступающим к работе в среде Enterprise Java, ведущим разработчикам, которые осуществляют переход с монолитной архитектуры к архитектурам на основе микросервисов, а также для архитекторов ПО, осваивающих методику предметно-ориентированного проектирования для создания приложений.

УДК 004.42Java
ББК 32.972

ISBN 978-1-4842-4542-2 (англ.)
ISBN 978-5-97060-872-2 (рус.)

Посвящается Тине и Майе

Содержание

Об авторе

Виджей Наир (Vijay Nair) в настоящее время является руководителем подразделения проектирования и разработки платформ для банковских приложений типа «программное обеспечение как услуга» (SaaS) компании Oracle. Как энтузиаст предметно-ориентированного проектирования и распределенных систем, он обладает 18-летним практическим опытом в области проектирования архитектуры, создания и реализации особо ответственных приложений непрерывного действия для сферы финансовых услуг по всему миру. Виджей Наир доступен для общения на собственном веб-сайте www.practicalddd.com или через «Твиттер» @FusionVJ. Виджей Наир живет в Маунтин Вью (Калифорния) вместе с женой и дочерью.

О техническом рецензенте

Мануэль Хордан (Джордан) **Элера** (Manuel Jordan Elera) – разработчик и исследователь-самоучка (автодидакт), которому очень нравится изучать и осваивать новые технологии на собственном практическом опыте (экспериментах) и создавать новые технологические комбинации. Мануэль является обладателем наград Springy Award – Community Champion и Spring Champion 2013. Несмотря на дефицит свободного времени, он читает Библию и играет на гитаре, сочиняет музыку. Мануэль известен под псевдонимом dr_pompeii. Он является техническим рецензентом многочисленных книг издательства Apress, в том числе Pro Spring, Fourth Edition (2014); Practical Spring LDAP (2013); Pro JPA 2, Second Edition (2013) и Pro Spring Security (2013). Можно прочесть его 13 подробных руководств по многим технологиям Spring, а также связаться с ним лично через его блог на сайте www.manueljordanelera.blogspot.com и наблюдать за его деятельностью в аккаунте «Твиттера» @dr_pompeii.

Благодарности

В первую очередь моя самая глубокая сердечная благодарность за то, что создание этой книги стало возможным, адресована гуру Jakarta EE (Enterprise Edition) Реза Рахману (Reza Rahman). Работая в компании Oracle, он создал инициативное направление Cargo Tracker как эскизный проект для Java EE Patterns на основе предметно-ориентированного проектирования (DDD). Я буду вечно благодарен ему за предоставленную мне возможность участвовать в этом проекте.

Благодарю приверженца предметно-ориентированного проектирования и энтузиаста использования программной среды Axon Framework Свапнила Сурве (Swapnil Surve), инженера-архитектора ПО из Феникса (Phoenix) за техническое рецензирование содержимого и предложения по главам 5 и 6. Отдельная благодарность Элларду Бьюзи (Allard Buijze) (создателю программной среды Axon Framework) за техническое рецензирование главы 6.

Спасибо всем сотрудникам компании Oracle, которые помогали написать эту книгу, моим руководителям (Викраму, Тиксу, Чету) и моей команде подчиненных, которые ежедневно обучают меня (Сурабу, Шрипаду, Хари, Павану, Дашарату и Махендрану).

Спасибо городу Маунтин Вью (Калифорния) за предоставление отдельной комнаты в потрясающей библиотеке, где я провел многие часы за написанием этой книги.

Если в семье появляется новорожденный, то человек в здравом уме вряд ли займется написанием книги. Поддержка моей семьи была колоссальной на протяжении всего периода работы над книгой, и я безмерно благодарен за это.

Спасибо моим братьям Гаутаму, Рохиту, Сумиту и Сачину, моей сестре Вините и ее мужу Мадху, моим детям Варуну и Арье, наконец моим родителям и родителям моей жены, за то, что они не пожалели своего времени и были с нами, помогали нам.

Последняя, но самая главная благодарность моей жене Тине. Она разрывалась между множеством дел, заботилась о новорожденном и обеспечивала мне возможность работать над книгой, прилагая нечеловеческие усилия и в прямом смысле жертвуя собой. Эта книга твоя в той же мере, что и моя. Спасибо тебе.

Введение

Предметно-ориентированное проектирование (Domain Driven Design) никогда не было настолько востребованным, как в современном мире разработки программного обеспечения. Концепции и шаблоны предметно-ориентированного проектирования помогают создавать правильно спроектированные приложения масштаба предприятия вне зависимости от того, являются ли эти приложения привычными монолитным программами или более современными приложениями на основе микросервисов.

Цель этой книги – снять покров мистики с концепций предметно-ориентированного проектирования, представив практические методы их реализации для более привычных устоявшихся монолитных приложений, а также для современных приложений, использующих микросервисы. С помощью реально существующего проекта приложения Cargo Tracker в книге подробно рассматриваются практические реализации разнообразных шаблонов предметно-ориентированного проектирования для обоих стилей приложений с применением различных инструментальных средств и рабочих программных сред из Enterprise Java Space (Jakarta EE, Eclipse MicroProfile, Spring Boot и Axon Framework). При таком подходе читатель получает полностью завершенную картину и возможность практического использования любого из этих инструментальных программных средств и сред в собственном процессе предметно-ориентированного проектирования.

Надеюсь, что чтение этой книги принесет вам пользу.

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте http://dmkpress.com/authors/publish_book/ или напишите в издательство: dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Скачивание исходного кода

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt Publishing очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1

Предметно-ориентированное проектирование

Предметно-ориентированное проектирование (Domain Driven Design) предлагает основательную, систематическую и полнофункциональную методику проектирования и разработки программного обеспечения. Эта методика предоставляет набор инструментальных средств и конкретных методов, которые помогают уменьшить сложность бизнес-процесса при сохранении основной (центральной) бизнес-модели как базовой сущности методики.

Предметно-ориентированное проектирование являлось предпочтительной методикой для обычных (в подавляющем большинстве монолитных) проектов в течение длительного времени, но с появлением архитектуры микросервисов концепции предметно-ориентированного проектирования стали в постоянно возрастающей степени применяться даже к этой новой парадигме архитектуры.

Эта книга разделена на две основные части.

Моделирование концепций предметно-ориентированного проектирования

Реализация предметно-ориентированного проектирования начинается с моделирования процесса для идентификации артефактов (поддоменов, ограниченных контекстов, моделей доменов, правил доменов), которые отображаются в концепции предметно-ориентированного проектирования. В нескольких первых главах книги приводится обобщенный обзор концепций предметно-ориентированного проектирования, затем описывается полный процесс моделирования для идентификации и документирования связанных с ним артефактов с подробным рассмотрением вариантов использования на примере конкретного реально существующего приложения.

Реализация концепций предметно-ориентированного проектирования

Далее в книге подробнейшим образом рассматриваются реализации этих концепций. Используя Enterprise Java как основную платформу, мы исследуем три различные реализации:

- на основе монолитной архитектуры с использованием платформы Java EE/Jakarta EE;
- на основе архитектуры микросервисов на платформе MicroProfile;
- на основе архитектуры микросервисов на платформе Spring.

Эти реализации охватывают три основные платформы, которые наиболее широко распространены в среде Enterprise Java и предоставляют полную дета-

лизацию практической реализации шаблонов предметно-ориентированного проектирования.

Концепции предметно-ориентированного проектирования

После ознакомления с основной целью этой книги можно перейти к рассмотрению основных концепций предметно-ориентированного проектирования.

Предметная область/бизнес-домен

Самой первой основной концепцией, с которой необходимо познакомиться, является идентификация предметной области (problem space), или бизнес-домена (business domain). Предметная область/бизнес-домен – это отправной пункт для всего процесса предметно-ориентированного проектирования, где определяется главная бизнес-задача, которую вы намерены решить с помощью предметно-ориентированного проектирования.

Рассмотрим эту концепцию более подробно, используя несколько практических примеров.

Сначала рассматривается пример из области автофинансирования, показанный на рис. 1.1. Деловая сфера автофинансирования предполагает управление автоматическими кредитами и арендами, т. е. вы, как провайдер услуг автофинансирования, должны предоставлять кредиты/аренды клиентам, обслуживать их, наконец при возникновении каких-либо проблем устранять их или возвращать предоставленные активы. В этом случае предметную область можно определить как управление автокредитами/автоарендами. Эту предметную область также можно обозначить термином «основной бизнес-домен» (core business domain) и «бизнес-задача» (business problem), которую вы намерены решить с использованием предметно-ориентированного проектирования.

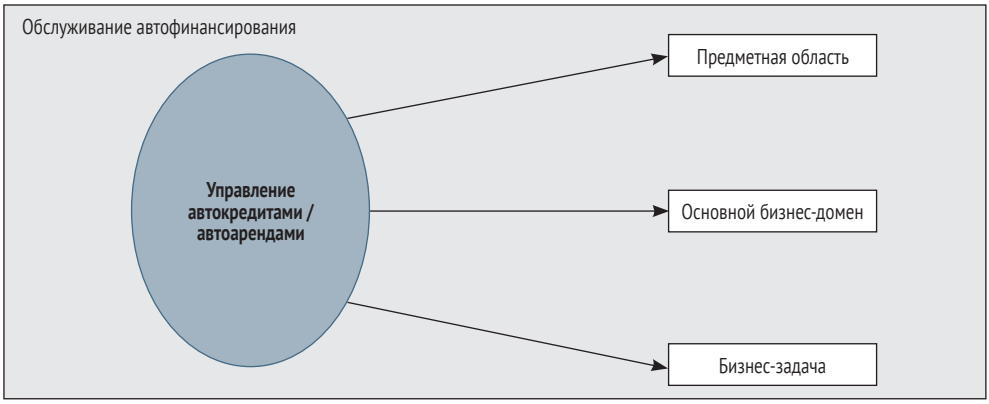

Рис. 1.1 ❖ Предметная область сферы обслуживания автофинансирования

Второй пример взят из банковской сферы. В отличие от первого примера здесь имеется не одна, а несколько предметных областей, в которых необходимо обеспечить решение с использованием предметно-ориентированного проектирования (см. рис. 1.2).

Рис. 1.2 ❖ Бизнес-домены (предметные области) в сфере обслуживания банком частных клиентов

От имени банка вы могли бы предложить обслуживание частных клиентов (рис. 1.2) в обобщенном случае или обслуживание корпоративных клиентов (рис. 1.3), если клиентом является предприятие или организация. В каждом рассматриваемом здесь варианте существует несколько предметных областей или основных бизнес-доменов.

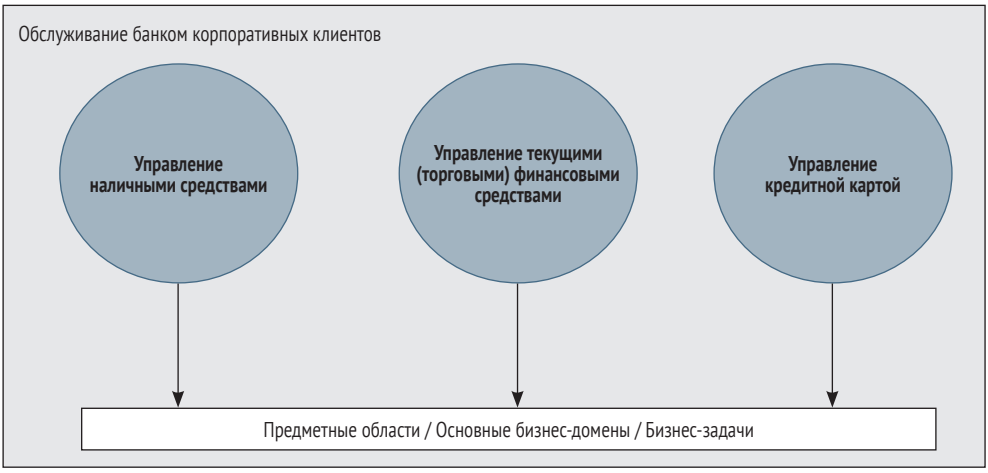

Рис. 1.3 ❖ Основные бизнес-домены в сфере обслуживания банком корпоративных клиентов

Предметные области/бизнес-домены всегда неизменно преобразовываются в основные проектные бизнес-утверждения, которые вы предлагаете как деловое предприятие.

Поддомены/ограниченные контексты

После определения основного бизнес-домена следующим шагом является разделение этого домена (предметной области) на поддомены (sub-domains). Определение поддоменов по существу означает разделение различных бизнес-возможностей вашего основного бизнес-домена на связанные друг с другом элементы функциональности бизнеса.

Снова обратимся к примеру из предметной области автофинансирования, которую можно разделить на три поддомена, как показано на рис. 1.4:

- поддомен «Первоначальные выплаты» – обеспечивает возможность выдачи новых автокредитов/предоставления автоматических аренд клиентам;
- поддомен «Обслуживание» – обеспечивает возможность обслуживания (например, ежемесячное выписывание счетов/сбор платежей) автокредитов/автоаренд;
- поддомен «Денежные сборы (инкассо)» – обеспечивает управление автокредитами/автоарендами, если возникают какие-либо проблемы (например, клиент не выполняет обязательства по платежам).

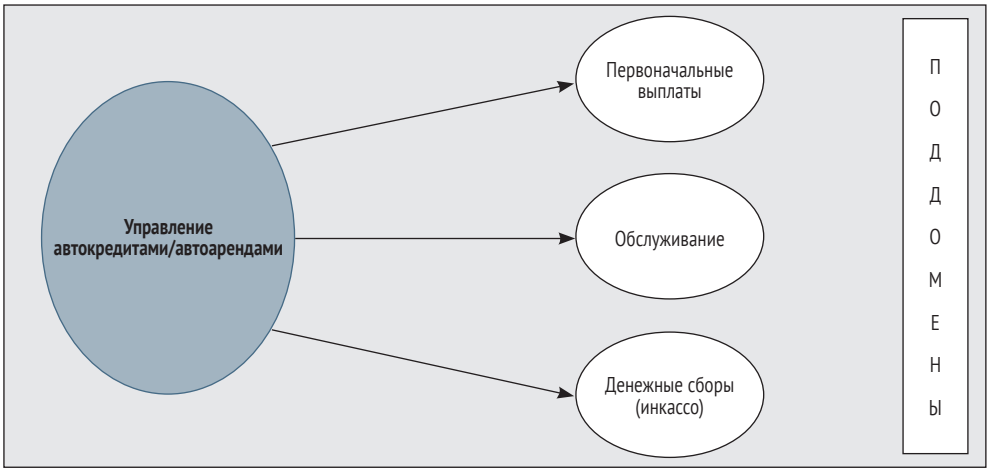

Рис. 1.4 ❖ Поддомены в предметной области «Управление автокредитами/автоарендами»

Очевидно, что поддомены определяются в терминах бизнес-возможностей (функций) вашей основной сферы деятельности, используемых повседневно.

На рис. 1.5 показан еще один пример определения поддоменов для одной из предметных областей сферы обслуживания банком частных клиентов – для бизнес-домена «Управление кредитной картой».

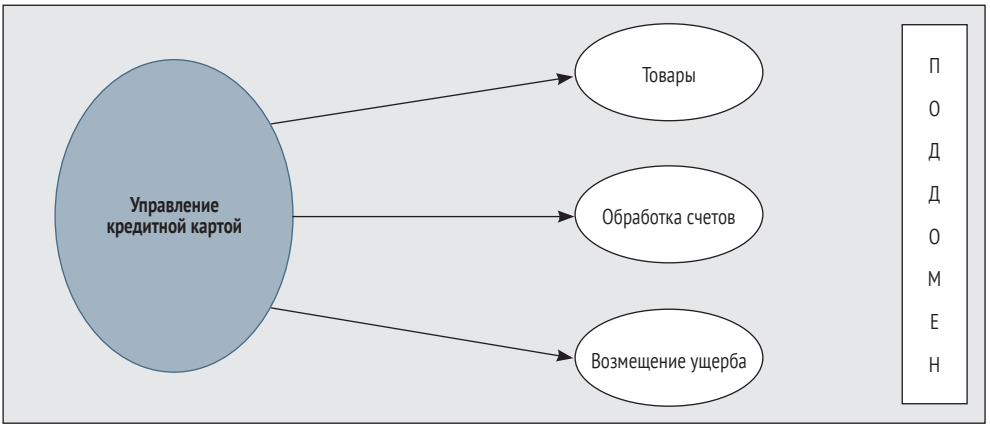

Рис. 1.5 ❖ Поддомены в предметной области «Управление кредитной картой»

Перечислим эти поддомены:

- «Товары» – определяет возможности по управлению всеми типами товаров, оплачиваемых кредитной картой;
- «Обработка счетов» – определяет возможности обработки счетов, оплачиваемых кредитной картой клиента;
- «Возмещение ущерба» – определяет возможности по управлению любыми типами исков, претензий и т. п. при возмещении ущерба по кредитной карте клиента.

И в этом случае действительные бизнес-возможности помогают точно определить поддомены.

Но что такое ограниченные контексты (bounded contexts)?

Напомним, что мы начали изучение процесса предметно-ориентированного проектирования с определения предметных областей (бизнес-доменов). Далее мы продолжили работу с бизнес-доменами, разделяя их на различные возможности (функции), чтобы определить соответствующие поддомены, отображающие эти функциональные возможности в конкретной деловой сфере.

Теперь необходимо приступить к созданию решений для соответствующих доменов/поддоменов, определенных ранее, т. е. необходимо переместиться из предметной области в область решений (solution space), где ограниченные контексты играют центральную роль.

Проще говоря, ограниченные контексты (bounded contexts) – это проектные решения для ранее определенных бизнес-доменов/поддоменов.

Процедура определения (идентификации) ограниченных контекстов руководствуется главным образом связностью, которая необходима внутри бизнес-домена и между установленными поддоменами.

Вернемся к первому примеру предметной области автофинансирования. Можно было бы выбрать единое решение для всей предметной области в целом, т. е. единый ограниченный контекст для всех поддоменов. Но можно определить ограниченные контексты, связанные с одним или с несколькими поддоменами.

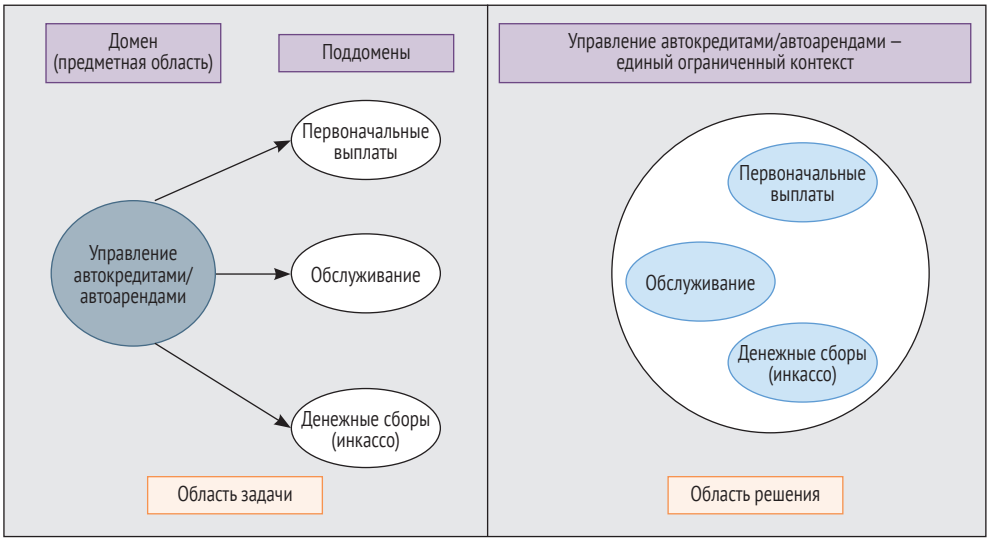

Рис. 1.6 ❖ Предметная область автофинансирования с решением в виде единого ограниченного контекста

Решением на рис. 1.6 для предметной области автофинансирования (автокредитов и аренд) является единый ограниченный контекст для всех поддоменов.

Другой подход – решения для различных поддоменов в предметной области автофинансирования как отдельные ограниченные контексты, как показано на рис. 1.7.

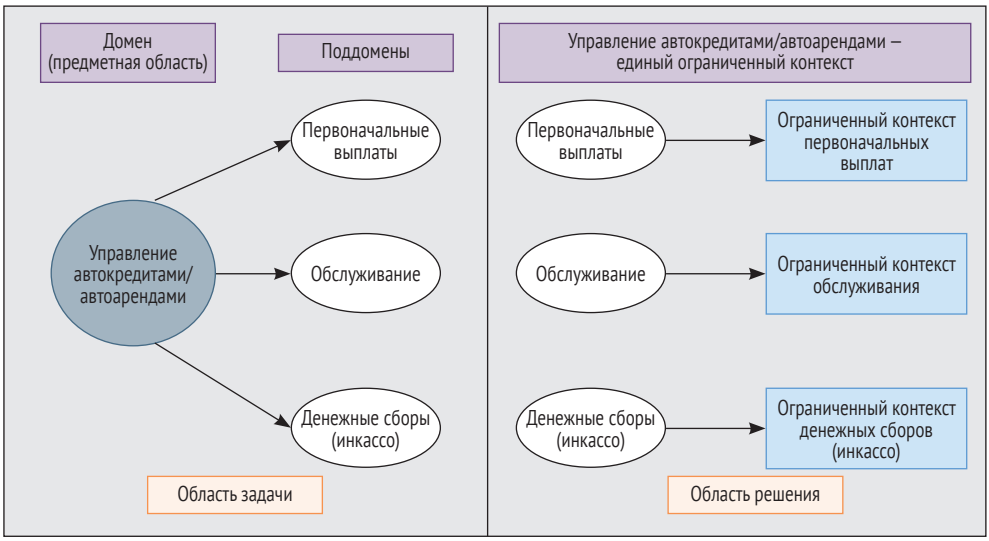

Рис. 1.7 ❖ Предметная область автофинансирования с решениями в виде отдельных ограниченных контекстов

При выборе конкретного подхода к развертыванию решения не существует никаких ограничений, поскольку ограниченный контекст интерпретируется как единый связный элемент. Можно использовать монолитное развертывание при подходе с несколькими ограниченными контекстами (единый веб-архив Web Archive [WAR] с несколькими JAR-файлами для каждого отдельного ограниченного контекста). Можно выбрать модель развертывания с использованием микросервисов, когда каждый ограниченный контекст представлен как отдельный контейнер. Также можно выбрать модель без сервера (serverless), в которой каждый ограниченный контекст развертывается как функция.

В последующих главах мы будем рассматривать каждый возможный тип модели развертывания как часть конкретных реализаций.

Модель предметной области

Теперь перейдем к самой важной и ответственной части процесса создания решения для предметной области – определению модели домена для ограниченного контекста. Если говорить кратко, то модель домена (domain model) – это реализация основной бизнес-логики внутри конкретно определенного ограниченного контекста.

На языке бизнеса это означает определение следующих элементов:

- бизнес-сущностей (бизнес-объектов);
- бизнес-правил;
- бизнес-потоков;
- бизнес-операций;
- бизнес-событий.

На техническом языке, принятом в мире предметно-ориентированного проектирования, элементы, подлежащие определению (идентификации), обозначаются следующим образом:

- агрегаты/сущности/объекты-значения;
- правила предметной области (домена);
- саги (sagas);
- команды/запросы;
- события.

Соответствие между приведенными выше двумя списками показано на рис. 1.8. В соответствии с показанной здесь схемой конструкции бизнес-языка отображаются в соответствующие конструкции технического языка предметно-ориентированного проектирования.

Поскольку мы будем подробно рассматривать все эти разнообразные концепции в последующих главах, необходимо кратко описать их здесь. Если эти концепции и элементы сейчас будут не вполне понятны, не беспокойтесь. В последующих главах вы получите полное представление об этих концепциях и элементах.

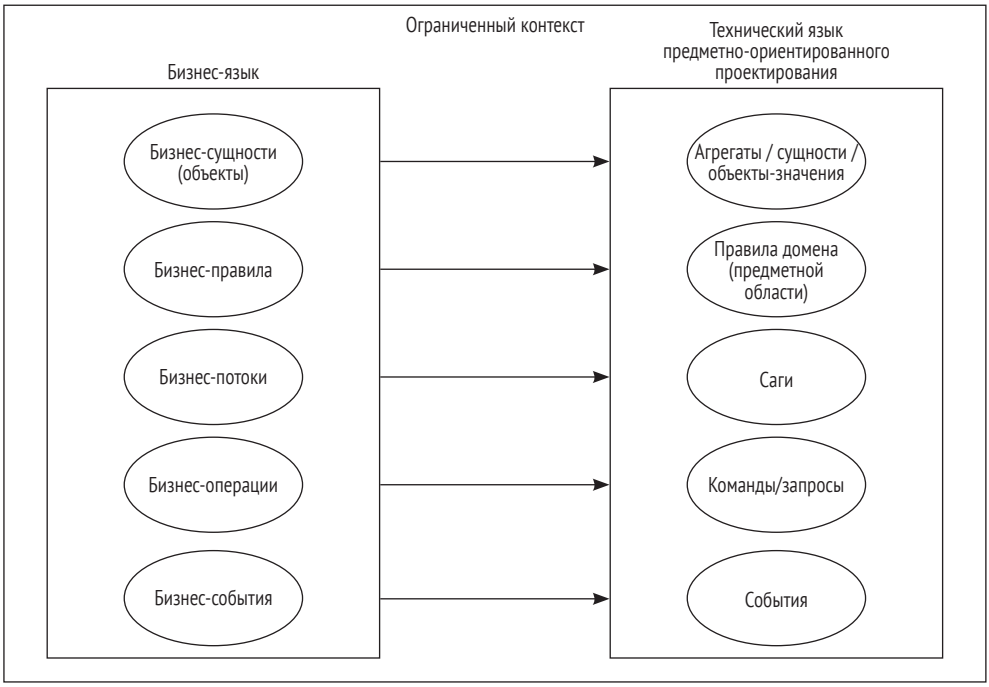

Рис. 1.8 ❖ Модель домена (предметной области) ограниченного контекста в терминах бизнес-логики и соответствующие им термины технического языка предметно-ориентированного проектирования

Агрегаты/объекты-сущности/объекты-значения

Агрегат (aggregate) (также называемый корневым агрегатом – root aggregate) – это центральный бизнес-объект в ограниченном контексте, который определяет область логической связности (целостности) внутри этого ограниченного контекста. Каждый аспект конкретного ограниченного контекста начинается и заканчивается внутри соответствующего корневого агрегата.

Агрегат = Самый главный идентификатор конкретного ограниченного контекста

Объекты-сущности (entity objects), или просто сущности (entity), обладают собственной идентичностью, но не могут существовать вне корневого агрегата, т. е. сущности создаются при создании корневого агрегата и уничтожаются при уничтожении корневого агрегата.

(Объекты)-сущности = Вторичные идентификаторы конкретного ограниченного контекста

Объекты-значения (value objects) не обладают собственной идентичностью и с легкостью заменяемы в любом экземпляре корневого агрегата или сущности.

В качестве примера рассмотрим ограниченный контекст первоначальных выплат в предметной области управления автоматическими кредитами/автоматическими арендами (рис. 1.9).

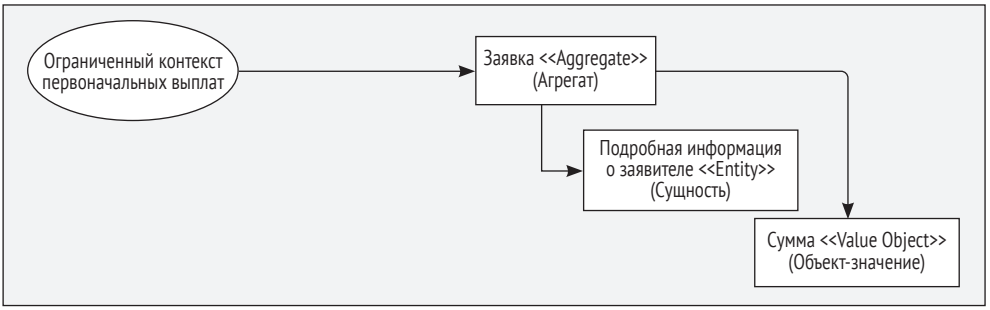

Рис. 1.9 ❖ Агрегаты/сущности/объекты-значения в ограниченном контексте первоначальных выплат

Агрегат заявки на кредит (Loan Application Aggregate) – это корневой агрегат в ограниченном контексте первоначальной выплаты. Без заявки на кредит невозможно существование чего-либо в этом ограниченном контексте, следовательно, кроме корневого агрегата в этом ограниченном контексте нет другого самого главного идентификатора.

Объект-сущность подробной информации о заявителе на кредит (Loan Applicant Details Entity Object) содержит все подробности о заявителе, подавшем заявку на кредит (личные данные, адрес и т. п.). Заявитель имеет собственный идентификатор (Applicant ID), но не может существовать без заявки на кредит (Loan Application), т. е. при создании заявки на кредит создается и подробная информация о заявителе. Разумеется, при удалении заявки на кредит удаляется и вся информация о заявителе.

Объект-значение сумма кредита (Loan Amount Value Object) – это сумма кредита для данной конкретной заявки. Она не имеет собственного идентификатора и может быть заменена в любом экземпляре агрегата заявки на кредит.

Рассматриваемое в последующих главах конкретное реально существующее приложение позволяет более подробно изучить все описанные выше концепции, поэтому, даже если вы сейчас поняли не все, не волнуйтесь. В настоящий момент мы просто отметим, что необходимо определить агрегаты, сущности и объекты-значения.

Правила предметной области

Правила предметной области (домена) – это, по существу, определения бизнес-правил. Правила предметной области, также моделируемые как объекты, способствуют выполнению агрегатом любого типа бизнес-логики в области соответствующего ограниченного контекста.

В рассматриваемом примере ограниченного контекста первоначальных выплат хорошим примером правила предметной области является бизнес-правило «Проверка состояния согласия заявителя» (State Applicant Compliance

Validation). Это правило, по существу, утверждает, что, в зависимости от «состояния» (state) заявки на кредит (например, CA (согласие получено), NY (согласие пока не получено)), могут выполняться дополнительные проверки выполнимости этой заявки.

Правило предметной области «Проверка состояния согласия заявителя» работает совместно с агрегатом «Заявка на кредит» (Loan Aggregate) для проверки (валидации) конкретной заявки на кредит на основе состояния при создании заявки, как показано на рис. 1.10.

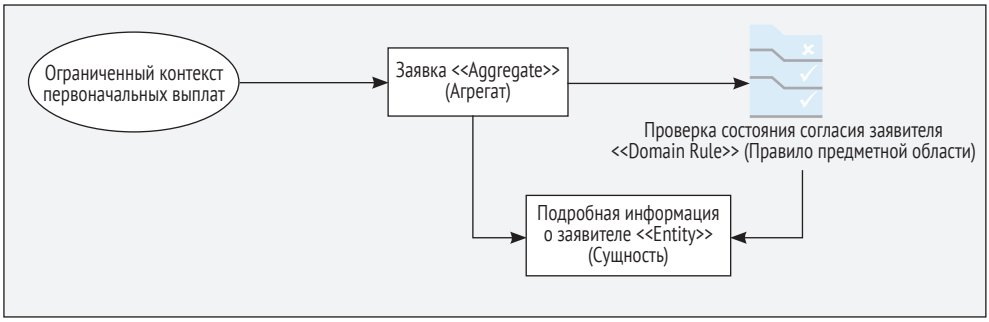

Рис. 1.10 ❖ Правила предметной области (домена) в ограниченном контексте первоначальных выплат

Команды/запросы

Команды (commands) и запросы (queries) представляют любой тип операций в ограниченном контексте, которые либо воздействуют на состояние агрегата/сущности, либо запрашивают состояние агрегата/сущности.

Как показано на рис. 1.11, некоторые примеры команд в ограниченном контексте первоначальных выплат могут включать команды «Открытие счета кредита» и «Изменение подробной информации о заявителе на кредит». Примерами запросов являются «Просмотр подробной информации по счету кредита» и «Просмотр подробной информации о заявителе на кредит».

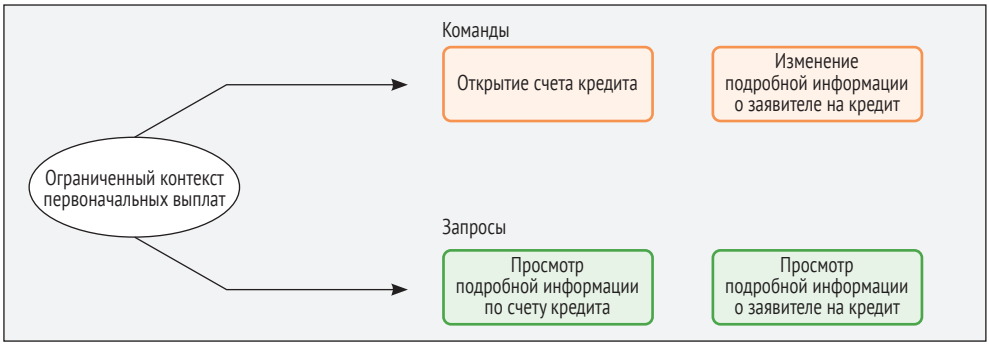

Рис. 1.11 ❖ Команды и запросы в ограниченном контексте первоначальных выплат

События

События (events) содержат любые типы изменения состояния как агрегата, так и сущности в ограниченном контексте. Это показано на рис. 1.12.

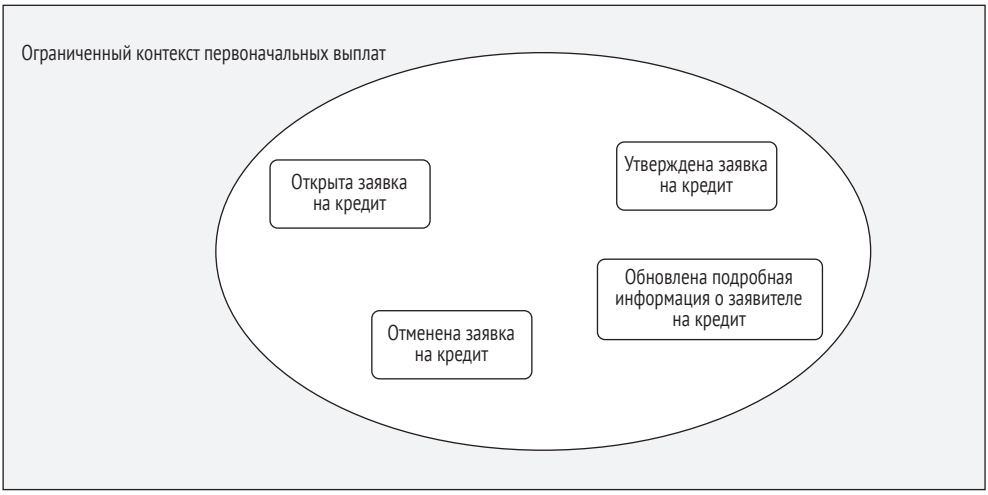

Рис. 1.12 ❖ События в ограниченном контексте первоначальных выплат

Саги

Последним элементом модели предметно-ориентированного проектирования является прогон любого типа бизнес-процессов/бизнес-потоков внутри конкретного бизнес-домена. Такой прогон в предметно-ориентированном проектировании обозначается термином сага (saga). Утверждается, что сага – это просто артефакт, который не ограничен одним ограниченным контекстом и может распространяться на несколько из них. В большинстве случаев сага охватывает множество ограниченных контекстов.

Ограниченный контекст и в особенности агрегат в нем действует как участник саги. Саги реагируют на многочисленные бизнес-события в различных ограниченных процессах и «выполняют оркестровку бизнес-процесса», координируя взаимодействия между всеми этими ограниченными контекстами.

Рассмотрим пример саги в уже знакомой нам предметной области автофинансирования – открытии кредитного счета.

Общая схема бизнес-процесса для открытия кредитного счета описана ниже.

1. Клиент передает заявку на кредит (Loan Application) в компанию X Auto Finance с целью приобретения нового автомобиля.
2. Компания X Auto Finance проверяет (выполняет валидацию) подробные данные этой заявки на кредит, чтобы предоставить наилучшие условия кредитования (Loan Product) клиенту.
3. Компания X Auto Finance либо утверждает заявку, либо отказывает клиенту в выдаче кредита.

4. Если заявка утверждена, то компания X Auto Finance представляет клиенту условия кредитования (Loan Product Terms), включая процентную ставку, срок владения (погашения) и т. д.
5. Клиент принимает условия кредитования.
6. Компания X Auto Finance окончательно утверждает заявку после согласования с клиентом.
7. Компания X Auto Finance открывает новый кредитный счет (Loan Account) для этого клиента.

Вполне очевидно, что этот бизнес-процесс включает несколько ограниченных контекстов, т. е. начинается с ограниченного контекста первоначальных выплат (утверждение заявки на кредит), заканчивается в ограниченном контексте обслуживания (открытие кредитного счета). Этот бизнес-процесс показан на рис. 1.13.

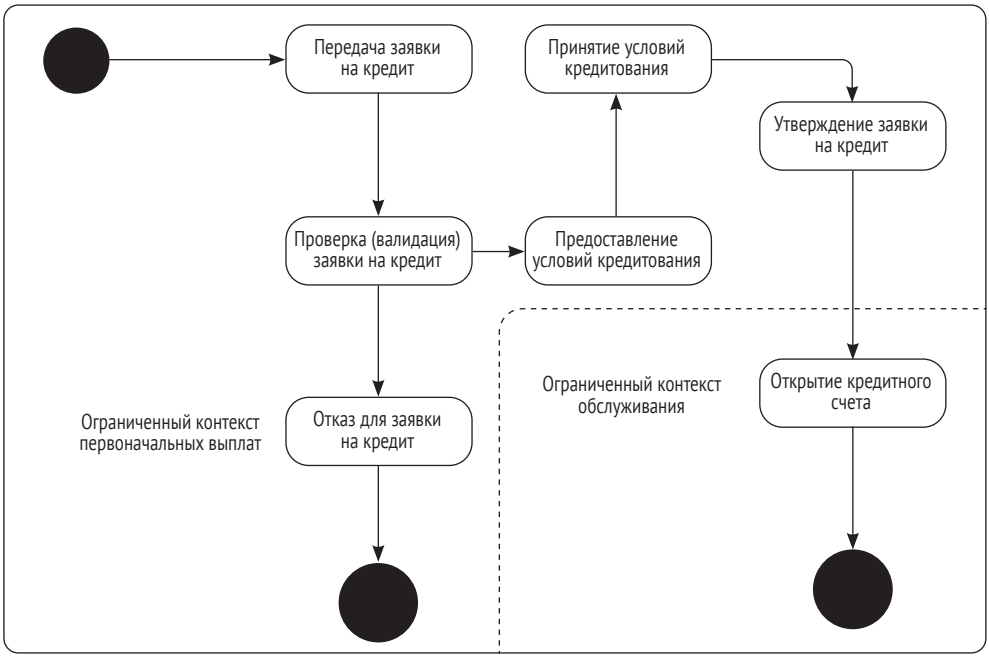

Рис. 1.13 ❖ Сага открытия кредитного счета

Теперь мы определили модель домена (предметной области) для рассматриваемого бизнес-домена и готовы для реализации этой модели.

Резюме

Краткое обобщение содержимого этой главы:

- процесс начинается с определения задачи основной предметной области или бизнес-домена, которую необходимо решить с использованием предметно-ориентированного проектирования;

- после определения предметная область разделяется на несколько бизнес-возможностей (функций) или поддоменов. Затем начинается перемещение в область решений посредством определения соответствующих ограниченных контекстов;
- завершающий этап – более глубокое погружение в область решений с определением модели предметной области (домена) для ранее установленного ограниченного контекста. На этом этапе выполняется идентификация (определение) агрегатов/операций/процессов и потоков в каждом ограниченном контексте.

Глава 2

Проект Cargo Tracker

Проект Cargo Tracker в этой книге будет главным реально существующим приложением, к которому мы будем обращаться при рассмотрении примеров реализации. Этот проект в течение длительного времени является своеобразным справочником по методикам и технологиям предметно-ориентированного проектирования, поэтому на протяжении всей книги мы будем рассматривать его практическую реализацию с использованием инструментальных средств, методов и функциональных возможностей, предоставляемых различными платформами Enterprise Java.

Приложение Cargo Tracker используется предприятиями и организациями, работающими в сфере доставки грузов (cargo). Проект предоставляет функциональные возможности по управлению полным жизненным циклом доставки грузов, включая заказ (booking), определение маршрута доставки (routing), отслеживание доставки (tracking) и транспортную обработку грузов, в том числе погрузочно-разгрузочные работы (handling). Приложение предназначено для использования бизнес-операторами, клиентами-заказчиками и службами обработки грузов (в портах и т. п.).

В этой главе мы проведем подготовительную работу для всех последующих вариантов реализации, определив в первую очередь специализированный для предметно-ориентированного проектирования процесс моделирования предметной области (домена). Основная цель этого процесса моделирования – точное определение набора артефактов предметно-ориентированного проектирования высокого и низкого уровня. Для артефактов высокого уровня требуется низкая степень реализации, т. е. это в большей степени концепции проектирования с минимумом требуемых физических артефактов. С другой стороны, для артефактов низкого уровня требуется высокая степень реализации, т. е. они представляют собой реальные физические артефакты рассматриваемой нами реализации.

Процесс моделирования предметной области (домена) применяется вне зависимости от выбора архитектуры на основе монолитной структуры или на основе микросервисов.

Основная предметная область (домен)

Чтобы начать работу в истинном духе предметно-ориентированного проектирования, в первую очередь необходимо определить, что нашей основной пред-

метной областью (доменом)/областью задачи является управление доставкой грузов, и рассматриваемое в этой книге приложение Cargo Tracker определяет именно эту предметную область/область задачи.

После определения основной предметной области (домена) определяются артефакты предметно-ориентированного проектирования для этой основной предметной области. Мы определяем четыре основных артефакта как часть рассматриваемого процесса:

- поддомены/ограниченные контексты основной предметной области (домена);
- модель предметной области (домена);
- саги предметной области (домена);
- сервисы модели предметной области (домена).

На рис. 2.1 показан процесс моделирования предметной области (домена).

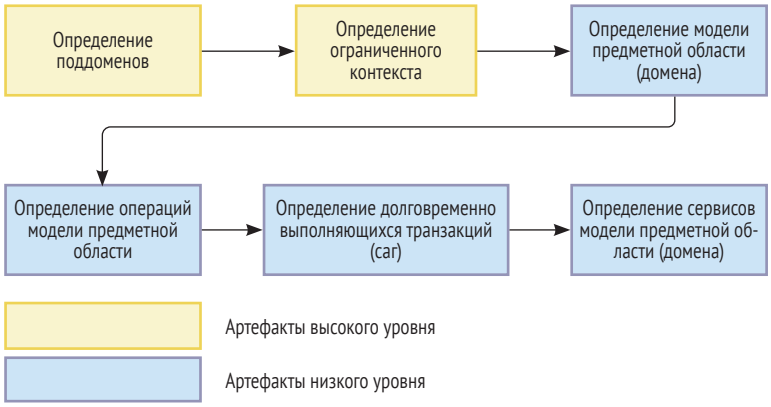

Рис. 2.1 ❖ Агрегаты в рассматриваемом ограниченном контексте

Проект Cargo Tracker: поддомены/ограниченные контексты

Для определения различных поддоменов в основной предметной области/области задачи Cargo Tracker домен разделяется на отдельные бизнес-области с классификацией каждой такой бизнес-области как поддомена (sub-domain).

В предметной области (домене) проекта Cargo Tracker определяются четыре основные бизнес-области:

- заказ (booking) – эта область включает все аспекты заказов на доставку грузов, в том числе следующие операции:
  - заказ на доставку грузов,
  - назначение маршрутов доставки грузов,
  - изменение условий доставки грузов (например, изменение пункта назначения заказанного груза),
  - отмена заказов на доставку грузов;

- определение маршрута доставки груза (routing) – эта область охватывает все аспекты планирования маршрута доставки, включая следующие операции:
  - оптимальное планирование упаковки и размещения грузов в транспортном средстве на основе соответствующей спецификации маршрута доставки,
  - обслуживание рейсов транспортных средств, которые будут доставлять грузы (например, добавление новых маршрутов доставки);
- транспортная обработка грузов, в том числе погрузочно-разгрузочные работы (handling) – поскольку грузы перемещаются по назначенным маршрутам, потребуется инспектирование/обработка в различных конечных пунктах их доставки. Эта область включает все операции, связанные с обработкой, в том числе и с погрузочно-разгрузочными работами доставляемых грузов;
- отслеживание доставки (tracking) – клиентам-заказчикам необходима полная, подробная и своевременная информация о грузах, доставку которых они заказали. Эта бизнес-область предоставляет такую возможность.

Каждую из описанных выше бизнес-областей можно классифицировать как поддомен в соответствии с парадигмой предметно-ориентированного проектирования. Определение поддоменов является частью определения области задачи, но, кроме того, необходимо определить и решения для этих поддоменов. Как мы уже видели в предыдущей главе, для этого используется концепция ограниченных контекстов. Ограниченные контексты (bounded contexts) – это проектные решения для конкретной рассматриваемой области задачи (предметной области), при этом каждый ограниченный контекст может иметь один или несколько поддоменов, связанных с ним.

Для всех реализаций, рассматриваемых в этой книге, предполагается, что каждый ограниченный контекст связан с единственным поддоменом.

Необходимость определения поддоменов не зависит от выбора стиля архитектуры, которому вы намерены следовать при создании приложения: монолитная архитектура или архитектура на основе микросервисов. Сущность определения поддоменов состоит в том, чтобы на протяжении всего процесса обеспечить четкое разделение основной предметной области на различные бизнес-области, независимые друг от друга, которые могут иметь собственный бизнес-язык, распознаваемый только в своей конкретной бизнес-области/поддомене.

На рис. 2.2 показаны различные поддомены основной предметной области проекта Cargo Tracker как модули в монолитной структуре, таким образом, и ограниченные контексты решаются в виде модулей.

На рис. 2.3 показаны различные поддомены основной предметной области проекта Cargo Tracker как отдельные микросервисы, таким образом, и ограниченные контексты решаются в виде микросервисов.

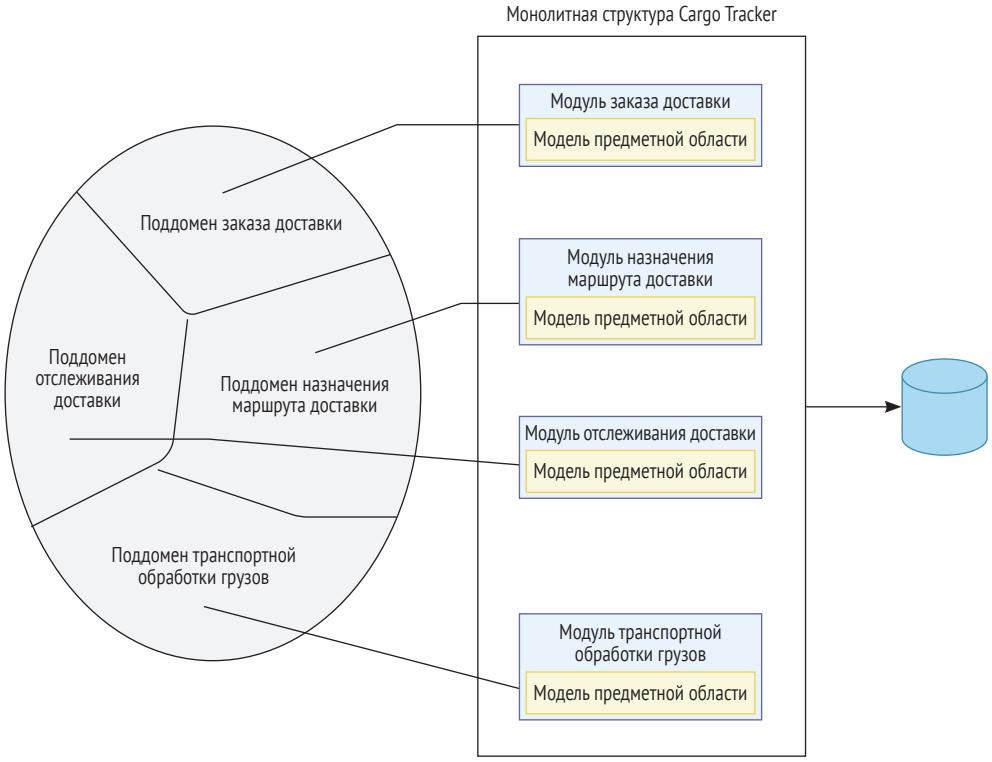

Рис. 2.2 ❖ Поддомены приложения Cargo Tracker как отдельные модули в монолитной структуре

Проектное решение определенных выше поддоменов выполняется через ограниченные контексты, развертываемые либо как модули в монолитной архитектуре, либо как отдельные микросервисы в архитектуре на основе микросервисов.

Резюмируя этот раздел, отметим, что, используя концепцию бизнес-областей, мы разделили основную предметную область (домен) на несколько поддоменов и определили ограниченные контексты как решения для этих поддоменов. Ограниченные контексты проектируются по-разному в зависимости от типа разрабатываемого решения. В монолитной архитектуре они реализуются как модули, в архитектуре на основе микросервисов – как отдельные микросервисы. Здесь важно отметить, что проектная реализация конкретных ограниченных контекстов основана на нашем первоначальном решении о связывании каждого ограниченного контекста с отдельным поддоменом. Это обобщенная практическая методика, необходимая в определенных случаях для решения с несколькими модулями в рамках одного ограниченного контекста в варианте монолитной архитектуры и для решения с несколькими микросервисами в рамках одного ограниченного контекста в варианте архитектуры на основе микросервисов. Ограниченный контекст – это окончательное решение.

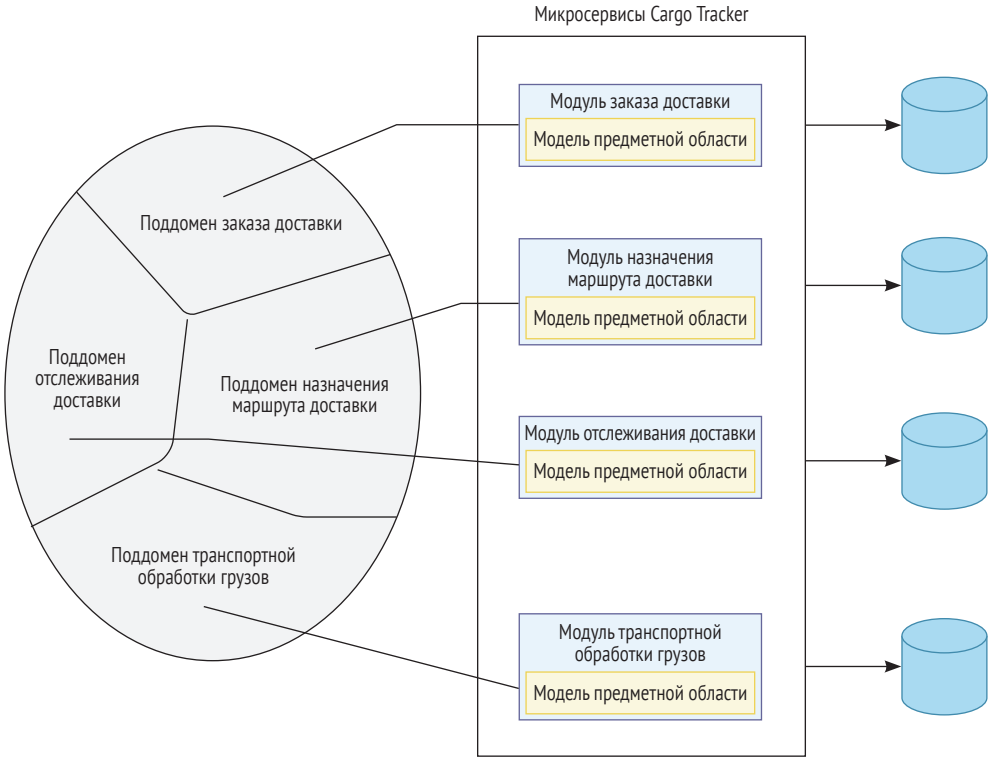

Рис. 2.3 ❖ Поддомены приложения Cargo Tracker как отдельные микросервисы

Следующий этап – определение модели предметной области (домена) для каждого ограниченного контекста.

Проект Cargo Tracker: модель предметной области (домена)

Модель предметной области (домена) для ограниченного контекста – это основополагающий элемент любой архитектуры на основе предметно-ориентированного проектирования, который используется для выражения бизнес-цели для конкретного ограниченного контекста. При определении модели предметной области определяется два основных набора артефактов:

- модель основной предметной области (домена) – агрегаты, идентификаторы агрегатов, сущности и объекты-значения;
- операции модели предметной области (домена) – команды, запросы и события.

Агрегаты

Наиболее важным основополагающим аспектом проектирования модели предметной области (домена) является определение агрегатов (aggregates) в ограниченном контексте. Агрегат определяет область логической целостности (связности) в конкретном ограниченном контексте, т. е. агрегат состоит из корневой сущности (root entity) и набора объектов-сущностей (entity objects) и объектов-значений (value objects). Агрегат можно рассматривать как единый элемент во время любой операции, которая обновляет состояние этого агрегата как единого целого. Агрегаты отвечают за сохранение всех состояний и бизнес-правил, связанных с соответствующим ограниченным контекстом.

На рис. 2.4 показаны агрегаты в ограниченных контекстах приложения Cargo Tracker.

Рис. 2.4 ❖ Агрегаты в ограниченных контекстах приложения Cargo Tracker

Определение агрегатов помогает установить область видимости (действия) каждого ограниченного контекста. Теперь необходимо определить идентификатор для каждого из агрегатов.

Идентификаторы агрегатов

Каждый агрегат должен быть недвусмысленно обозначен с использованием идентификатора агрегата (aggregate identifier). Идентификатор агрегата реализуется с помощью бизнес-ключа (business key). Для реализации приложения Cargo Tracker на рис. 2.5 показаны бизнес-ключи для ранее определенных агрегатов.

Каждый ограниченный контекст выражает логику своей предметной области через набор связей собственного агрегата, реализованный с использованием объектов-сущностей (entity objects) и объектов-значений (value objects). Теперь мы должны определить их в рассматриваемом приложении Cargo Tracker.

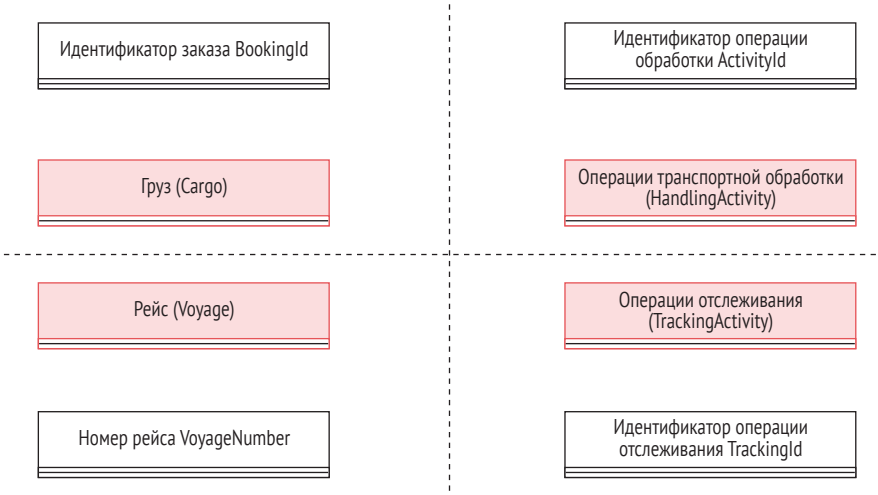

Рис. 2.5 ❖ Идентификаторы агрегатов, определенные с использованием бизнес-ключей

Сущности

Сущности (entities) в ограниченном контексте обладают собственной идентичностью, но не могут существовать без агрегата. Кроме того, сущности в агрегате заменить невозможно. Рассмотрим пример, который поможет определить правила для идентификации сущностей.

В агрегате Cargo (Груз) (ограниченный контекст заказа перевозки груза) как части процесса заказа, служащий, регистрирующий заказы, должен определить исходное состояние (источник, исходный пункт) заказываемого груза. Это отображается как объект-сущность, т. е. локация (Location), явно имеющая собственную идентичность, но ее существование невозможно само по себе, без агрегата Cargo (Груз).

На рис. 2.6 показаны объекты-сущности в ранее определенных ограниченных контекстах. Простое практическое правило определения сущностей: необходимо, чтобы сущности имели собственную идентичность, но их невозможно заменить в соответствующем агрегате.

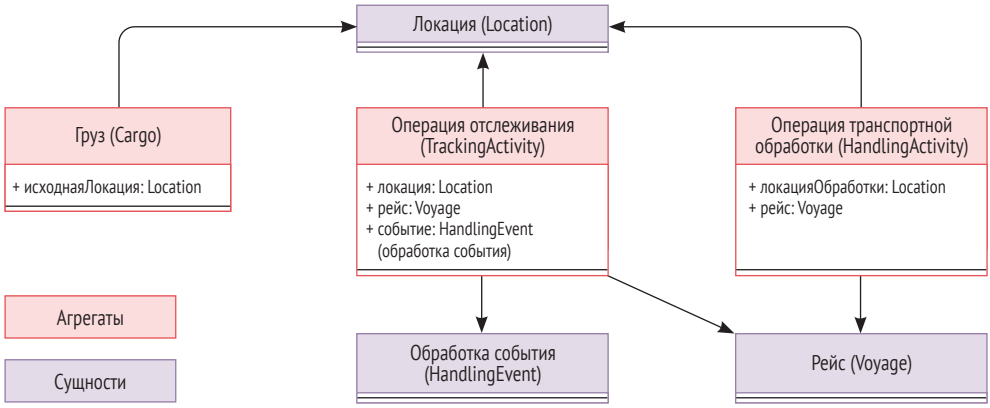

Рис. 2.6 ❖ Пример определенных сущностей

Объекты-значения

Объекты-значения (Value Objects) в ограниченном контексте не имеют собственной идентичности, и их можно заменить в любом экземпляре агрегата.

Рассмотрим пример, который поможет установить правила определения объектов-значений.

Агрегат Cargo (Груз) содержит следующие объекты-значения:

- заказанное количество (объем, масса и т. п.) груза (Booking Amount);
- спецификацию маршрута доставки (Route specification), исходную локацию (пункт отправки), целевую локацию (пункт доставки), предельный срок прибытия груза в пункт доставки);
- план маршрута (Itinerary), назначаемый для этого груза на основе спецификации маршрута доставки. План маршрута состоит из нескольких этапов пути (Legs), по которым может быть направлен груз для достижения пункта доставки;
- ход процесса доставки (Delivery Progress) назначается для груза вместе со спецификацией и планом маршрута. Ход процесса доставки предоставляет следующую подробную информацию: состояние маршрута (Routing Status), состояние транспорта (Transport Status), текущий рейс доставки груза (Current Voyage of the cargo), последняя известная локация, в которой находится груз (Last Known Location of the cargo), следующая ожидаемая операция (Next Expected Activity), последняя операция, выполненная с грузом (Last Activity that occurred on the cargo).

Теперь выполним проход по различным вариантам развития событий (сценариям) и дадим обоснование, почему перечисленные выше элементы являются объектами-значениями, а не сущностями, – потому что это весьма важное решение при моделировании предметной области (домена):

- при заказе нового груза ему назначается новая спецификация маршрута и пустой план маршрута, но не ход процесса доставки;
- как только для груза назначается конкретный план маршрута доставки, пустой план маршрута заменяется назначенным планом маршрута доставки этого груза;

- поскольку движение груза через несколько пунктов (портов и т. п.) является частью его плана маршрута, ход процесса доставки обновляется и заменяется в агрегате Cargo (Груз);
- наконец, если заказчик меняет пункт доставки или предельный срок прибытия груза, то изменяется спецификация маршрута, назначается новый план маршрута, пересчитывается срок доставки и изменяется заказанное количество груза.

Все рассмотренные выше элементы не обладают собственной идентичностью, и их можно заменять в агрегате Cargo (Груз), следовательно, они моделируются как объекты-значения. Это и есть простое практическое правило определения объектов-значений.

На рис. 2.7 показана полная диаграмма класса для агрегата Cargo (Груз) после добавления объектов-значений.

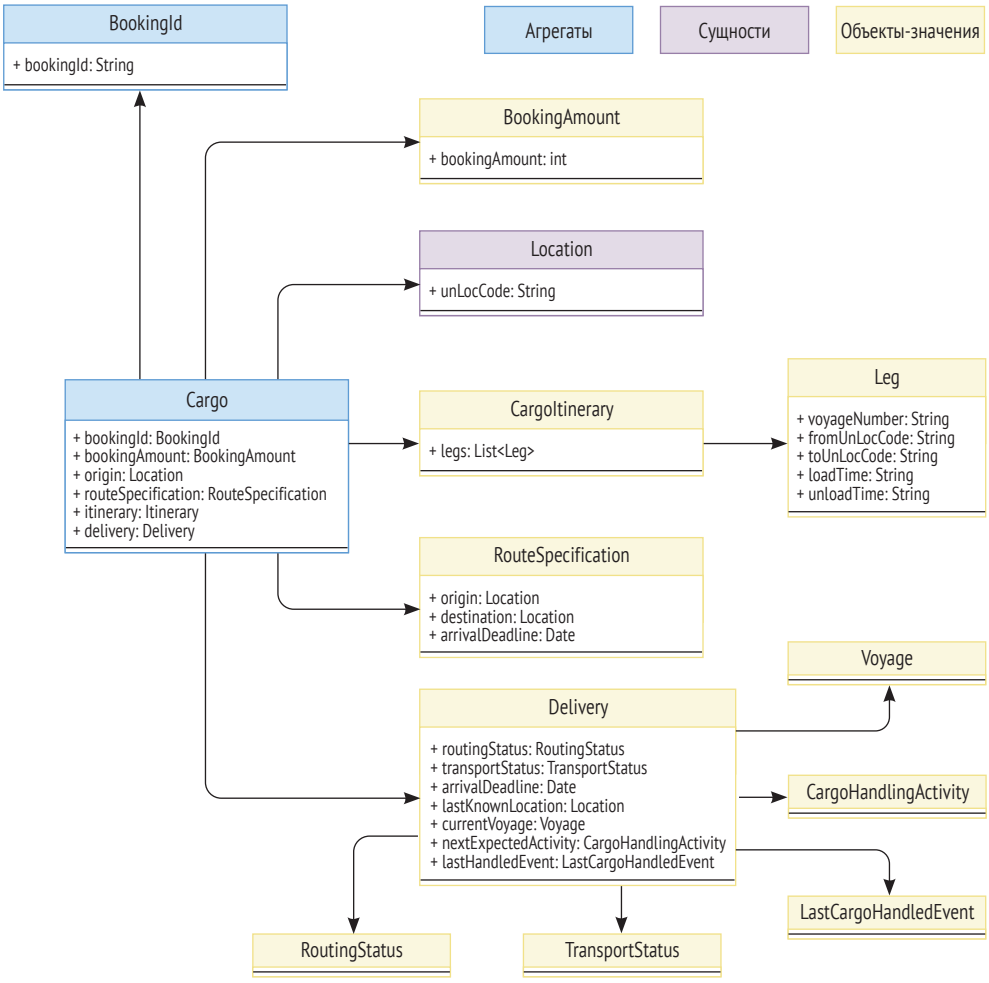

Рис. 2.7 ❖ Диаграмма класса агрегата Cargo (Груз)

Рассмотрим сокращенные диаграммы классов для других агрегатов, начиная с операций транспортной обработки грузов HandlingActivity (рис. 2.8).

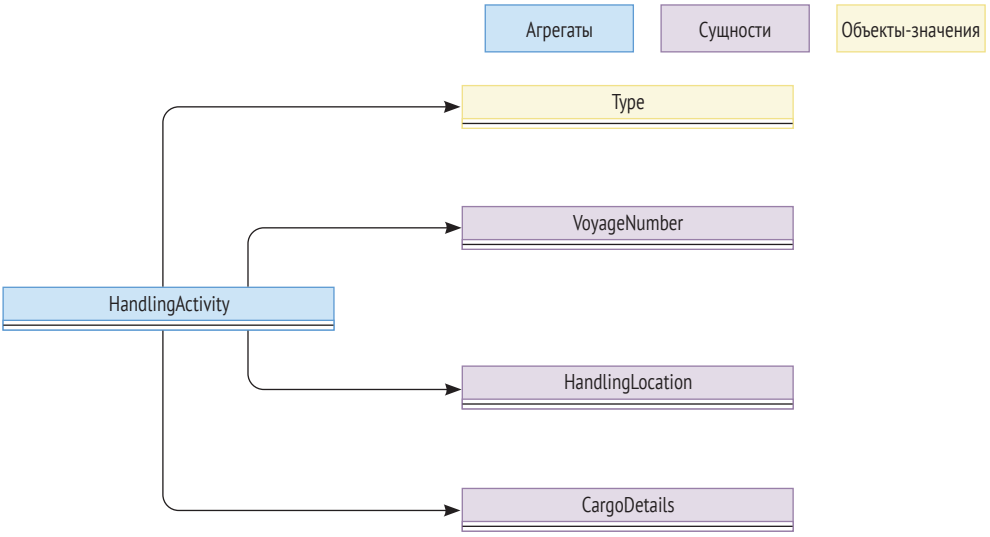

Рис. 2.8 ❖ Диаграмма класса HandlingActivity

На рис. 2.9 показан агрегат Voyage (Рейс).

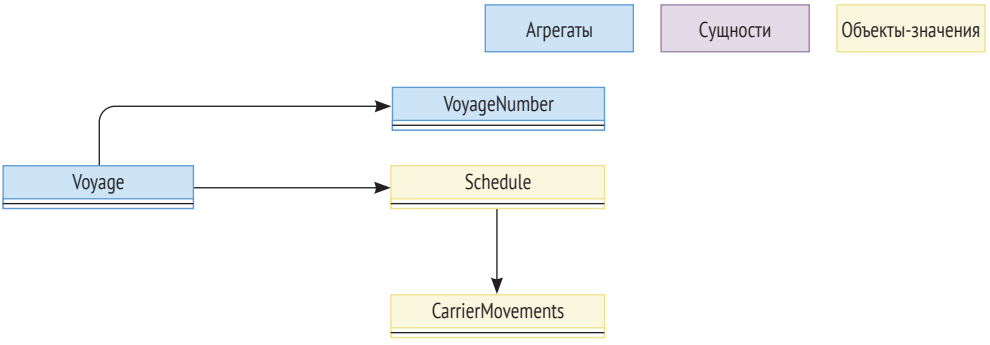

Рис. 2.9 ❖ Диаграмма класса для агрегата Voyage

Наконец, на рис. 2.10 изображен агрегат TrackingActivity (Операции отслеживания).

> **ПРИМЕЧАНИЕ** Исходный код для этой книги содержит реализацию модели основной предметной области (основного домена), демонстрируемую с разделением на пакеты. Чтобы получить четкое представление о типах объектов в модели основной предметной области (основного домена), вы можете тщательно изучить этот исходный код в репозитории GitHub www.github.com/après/practical-ddd-in-enterprise-java.

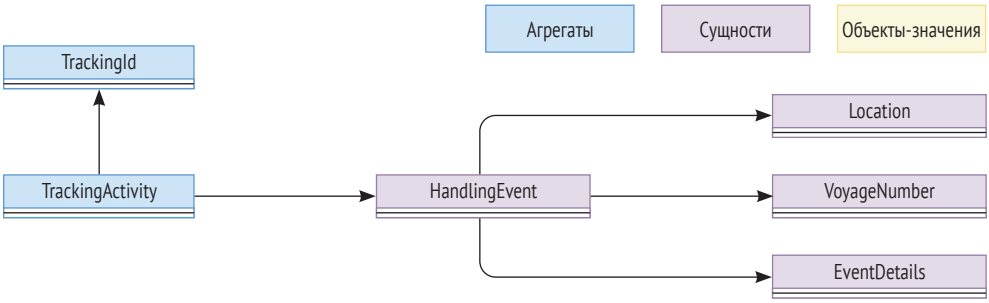

Рис. 2.10 ❖ Диаграмма класса для агрегата TrackingActivity

Проект Cargo Tracker: операции модели предметной области (домена)

Мы рассмотрели ограниченные контексты приложения Cargo Tracker и проработали модель основной предметной области (домена) для каждого ограниченного контекста. Следующий этап – определение операций модели предметной области (домена), которые должны выполняться в любом ограниченном контексте.

Операциями в любом ограниченном контексте могут быть:

- команды (commands), которые требуют изменения состояния в ограниченном контексте;
- запросы (queries), которые запрашивают состояние ограниченного контекста;
- события (events), которые сообщают об изменении состояния ограниченного контекста.

На рис. 2.11 показаны обобщенные операции в ограниченном контексте.

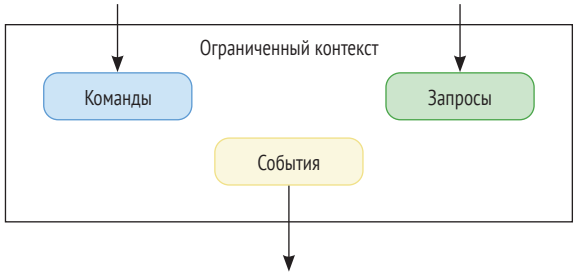

Рис. 2.11 ❖ Обобщенные операции в ограниченном контексте

На рис. 2.12 показаны операции модели предметной области (домена) для ограниченных контекстов приложения Cargo Tracker.

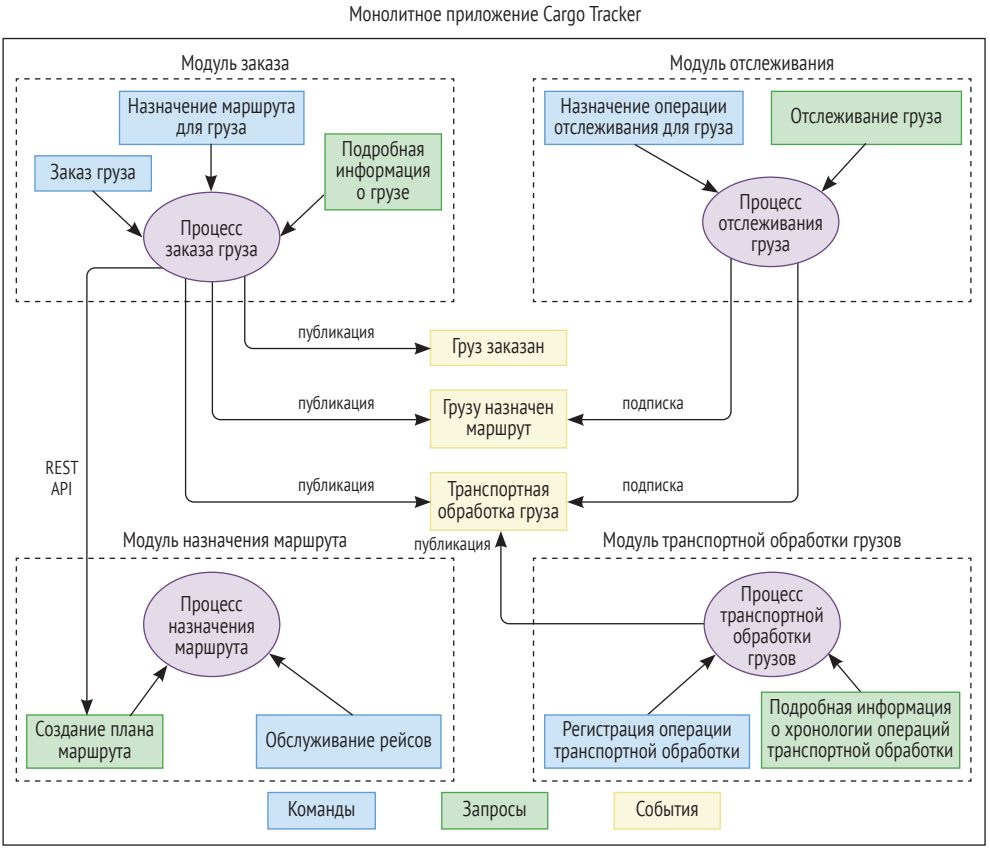

Рис. 2.12 ❖ Операции модели предметной области (домена) для ограниченных контекстов приложения Cargo Tracker

Саги

Саги (sagas) используются в основном в тех случаях, когда для разработки приложений применяется архитектура микросервисов. Распределенная сущность приложений на основе микросервисов требует реализации механизма для поддержания логической целостности данных в тех вариантах использования, при которых в работу вовлечено сразу несколько микросервисов. Саги помогают реализовать такой подход. Саги могут быть реализованы двумя способами: хореография событий (Event Choreography) или оркестровка событий (Event Orchestration):

- реализация саги на основе хореографии событий проста и понятна в том смысле, что микросервисы, участвующие в конкретной саге, будут напрямую активизироваться и подписываться на события;
- с другой стороны, в саге на основе оркестровки событий координация жизненного цикла выполняется через центральный компонент. Этот центральный компонент отвечает за создание саги, за регулирование

рабочего потока, проходящего через различные ограниченные контексты, участвующие в саге, наконец, за корректное завершение самой саги.

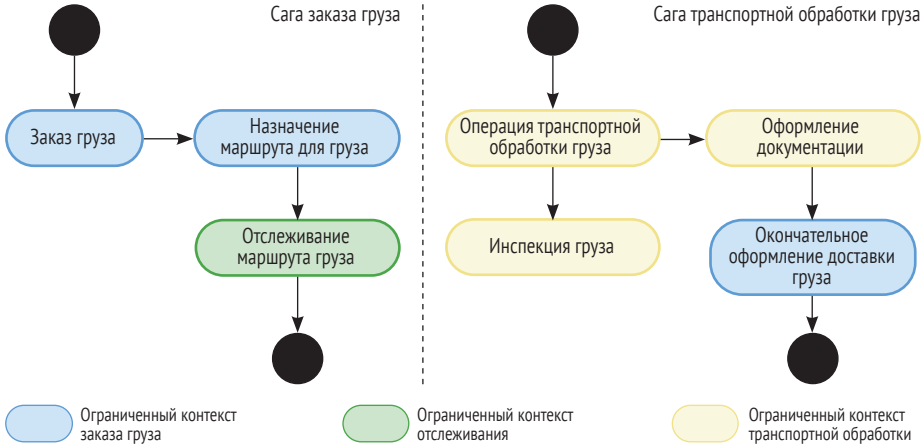

Рис. 2.13 ❖ Саги в приложении Cargo Tracker

На рис. 2.13 показано два примера саг в приложении Cargo Tracker.

Сага заказа груза включает бизнес-операции заказа груза, назначения маршрута доставки груза и отслеживания маршрута доставки груза. Она начинается с идентификации заказанного груза, затем назначается маршрут его доставки, и заканчивается сага присваиванием идентификатора отслеживания заказанному грузу. Этот идентификатор отслеживания используется клиентами-заказчиками для отслеживания движения груза по маршруту.

Сага транспортной обработки груза включает бизнес-операции транспортной обработки, инспекции, подтверждение транспортной обработки и оплаты груза. Она начинается с транспортной обработки (например, погрузки-разгрузки) в пунктах (портах), через которые проходит рейс и подтверждается клиентом-заказчиком в конечном пункте доставки, а завершается окончательным оформлением доставки груза (например, требованием выплаты штрафа за доставку с опозданием).

Обе эти саги проходят через несколько ограниченных контекстов/микросервисов, поэтому необходимо сохранение целостности транзакций при проходе через все ограниченные контексты и при завершении каждой саги.

Сервисы модели предметной области

Сервисы модели предметной области (домена) используются по двум основным причинам. Первая причина: обеспечение доступности модели предметной области ограниченного контекста для внешних участников процесса через правильно определенные интерфейсы. Вторая причина: обеспечение взаимодействия с внешними участниками процесса посредством надежного постоянного сохранения состояния ограниченного контекста в хранилищах данных (базах данных), публикации событий, изменяющих состояние ограниченного

контекста, во внешних брокерах сообщений (Message Brokers) или обмен информацией с другими ограниченными контекстами.

Для любого ограниченного контекста существует три типа сервисов модели предметной области (домена):

- входящие сервисы (inbound services), в которых реализуются четко определенные интерфейсы, позволяющие внешним участникам процесса взаимодействовать с моделью предметной области (домена);
- исходящие сервисы (outbound services), в которых реализуются все взаимодействия с внешними репозиториями и другими ограниченными контекстами;
- сервисы приложения (application services), действующие как функции внешнего уровня между моделью предметной области (домена) и входящими и исходящими сервисами.

На рис. 2.14 показан набор сервисов предметной области (домена) в монолитном приложении Cargo Tracker.

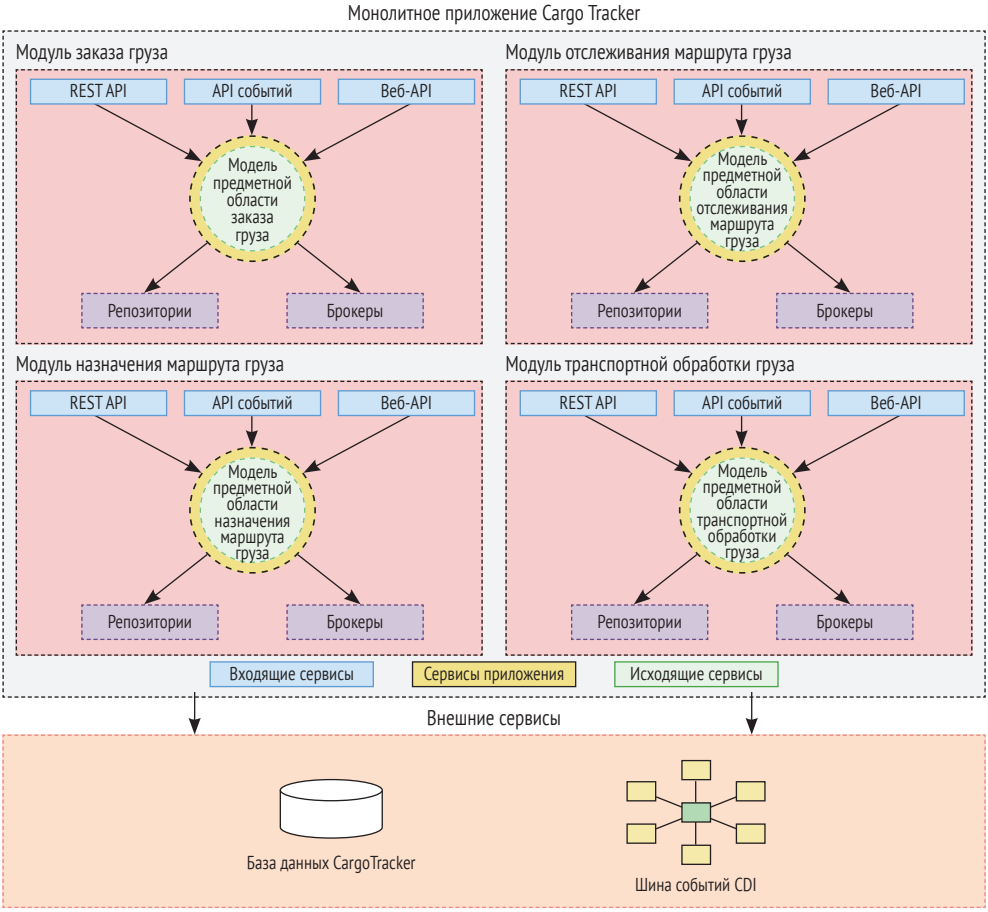

Рис. 2.14 ❖ Сервисы предметной области (домена) в монолитном приложении Cargo Tracker

На рис. 2.15 показан набор сервисов предметной области (домена) в приложении Cargo Tracker на основе микросервисов. В отличие от монолитного приложения микросервисы не предоставляют собственный встроенный веб-интерфейс.

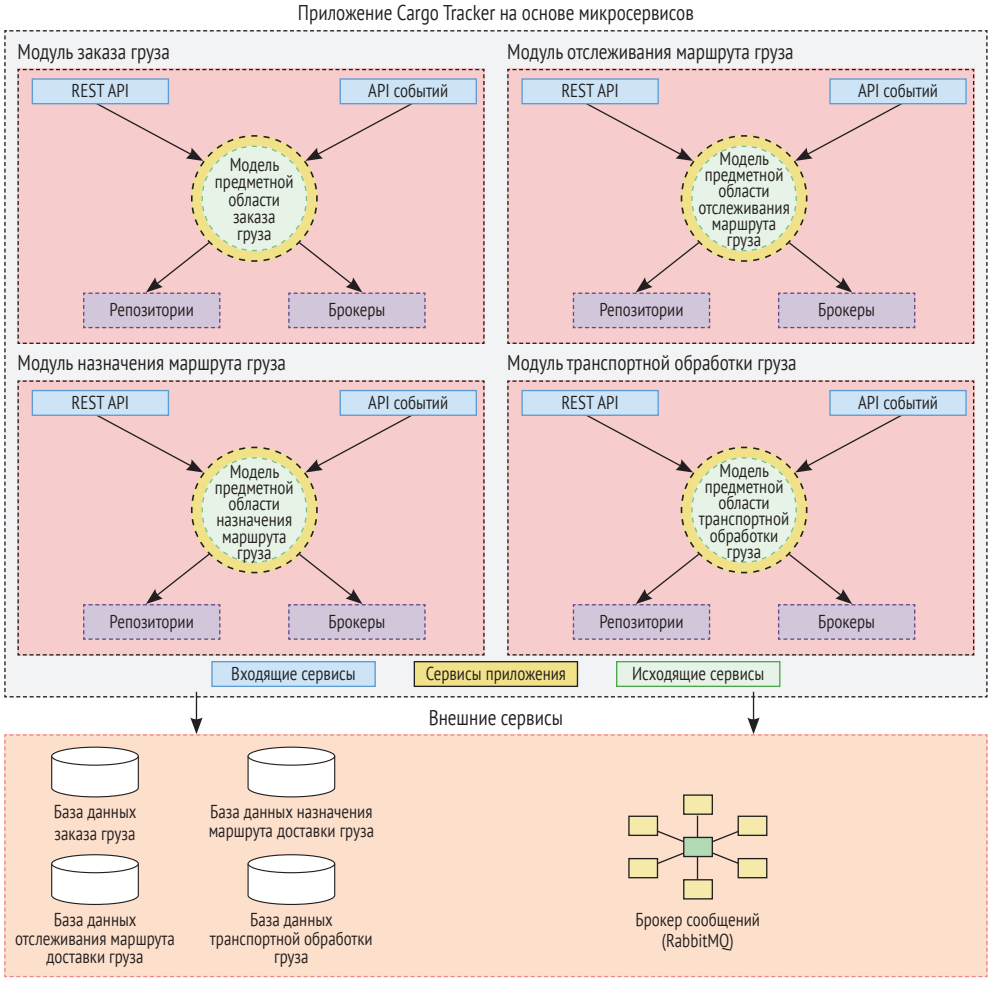

Рис. 2.15 ❖ Сервисы предметной области (домена) в приложении Cargo Tracker на основе микросервисов

Проектирование сервисов модели предметной области (домена)

Как спроектировать эти сервисы? Каким архитектурным шаблоном можно воспользоваться для реализации этих сервисов поддержки?

Шестиугольный или гексагональный архитектурный шаблон – превосходный выбор, который поможет нам смоделировать/спроектировать и реализо-

вать сервисы поддержки модели предметной области (домена). На рис. 2.16 показан гексагональный архитектурный шаблон.

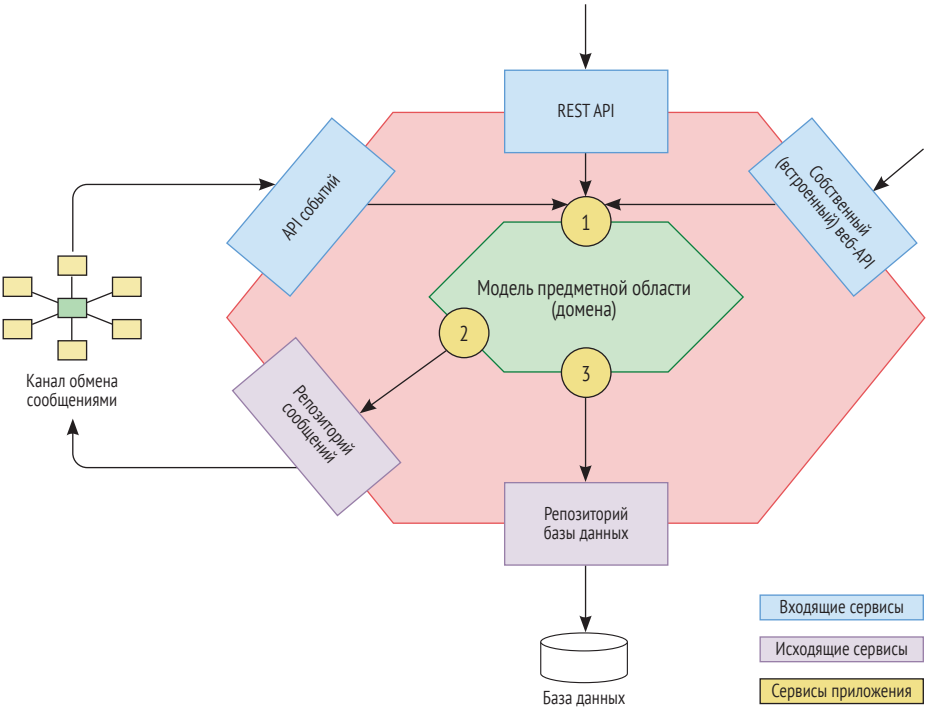

Рис. 2.16 ❖ Гексагональный архитектурный шаблон

Гексагональная архитектура использует концепцию портов (ports) и адаптеров (adaptors) для реализации сервисов модели предметной области (домена). Ниже приводится краткое описание этой концепции.

Порт в гексагональном архитектурном шаблоне может быть либо входящим, либо исходящим:

- входящий порт (inbound port) предоставляет интерфейс для бизнес-операций принятой модели предметной области (домена). Обычно это реализовано через сервисы приложения (см. элемент (1) на рис. 2.16);
- исходящий порт (outbound port) предоставляет интерфейс для технических операций, требуемых для принятой модели предметной области (домена). Модель предметной области (домена) использует эти интерфейсы для сохранения или публикации любого типа состояния из поддомена (см. элементы (2) и (3) на рис. 2.16).

Адаптер в гексагональном архитектурном шаблоне может быть либо входящим, либо исходящим:

- входящий адаптер (inbound adaptor) использует входящий порт для предоставления внешним клиентам возможностей использования модели предметной области (домена). Это реализуется через REST API, собственные встроенные веб-API или через API событий;

- исходящий адаптер (outbound adaptor) – это реализация исходящего порта для конкретного репозитория. Более подробно исходящие адаптеры будут описаны в следующих главах книги.

Резюмируя все сказанное выше, отметим, что для модели предметной области (домена) необходим набор поддерживающих сервисов, также известный под названием «сервисы модели предметной области (домена)». Эти поддерживающие сервисы позволяют внешним клиентам пользоваться принятой моделью предметной области (домена) и в то же время позволяют и модели предметной области сохранять и публиковать состояния поддомена (поддоменов) в нескольких репозиториях.

Эти поддерживающие сервисы смоделированы с использованием гексагонального архитектурного шаблона, в котором они отображаются либо как входящий/исходящий порт, либо как входящий/исходящий адаптер. Гексагональный архитектурный шаблон позволяет обеспечить независимость модели предметной области (домена) от этих поддерживающих сервисов.

На этом завершается рассматриваемый здесь конкретный процесс предметно-ориентированного проектирования. Мы разработали поддомены/ограниченные контексты для области задачи, подробно определили модель предметной области (домена) для каждого ограниченного контекста, а также операции предметной области (домена), выполняемые в каждом ограниченном контексте. Наконец, мы сформировали поддерживающие сервисы, необходимые для модели предметной области (домена).

Выполнение этого процесса проектирования не зависит от выбора монолитной архитектуры или архитектуры на основе микросервисов. Мы расширим этот процесс проектирования и более подробно рассмотрим его, когда приступим к реализации определенных ранее артефактов предметно-ориентированного проектирования, используя для этого инструментальные средства и методы, доступные в рабочей среде Enterprise Java.

Проект Cargo Tracker: реализации с использованием предметно-ориентированного проектирования

В следующих главах будет подробно рассматриваться реализация приложения Cargo Tracker для артефактов предметно-ориентированного проектирования, определенных к настоящему моменту.

Как часть этих реализаций мы будем проектировать и разрабатывать версии приложения Cargo Tracker:

- монолитную версию на основе предметно-ориентированного проектирования с использованием платформы Jakarta EE;
- версию с применением микросервисов на основе предметно-ориентированного проектирования с использованием платформы Eclipse MicroProfile;
- версию с применением микросервисов на основе предметно-ориентированного проектирования с использованием платформы Spring Boot;

- версию с применением микросервисов на основе предметно-ориентированного проектирования с использованием рабочей программной среды Axon Framework.

В следующей главе мы рассмотрим первый вариант реализации приложения Cargo Tracker как монолитной программы.

Резюме

Краткое обобщение содержимого этой главы:

- был выполнен обзор реально существующего приложения Cargo Tracker и определены поддомены/ограниченные контексты для этого приложения;
- сформирована модель основной предметной области (основного домена) приложения Cargo Tracker, включая определение агрегатов, сущностей и объектов-значений. Также определены операции модели предметной области (домена) и саги, связанные с приложением Cargo Tracker;
- глава завершается определением сервисов модели предметной области (домена), требуемых для конкретной модели предметной области (домена) приложения Cargo Tracker с использованием гексагонального (шестиугольного) архитектурного шаблона.

Глава 3

Проект Cargo Tracker: Jakarta EE

Мы подробно описали полный процесс моделирования различных артефактов предметно-ориентированного проектирования для любого приложения и теперь применим его на практике для конкретного приложения Cargo Tracker.

Краткий обзор содержимого предыдущих глав:

- процесс наблюдения за доставляемым грузом был определен как область задачи, или предметная область, или основной домен, а приложение Cargo Tracker – как решение для этой области задачи;
- были определены различные поддомены/ограниченные контексты для приложения Cargo Tracker;
- была подробно описана модель предметной области (домена) для каждого из определенных ограниченных контекстов, в том числе определение агрегатов, сущностей, объектов-значений и правил предметной области (домена);
- были определены поддерживающие сервисы предметной области (домена), требуемые в этих ограниченных контекстах;
- также были определены разнообразные операции в этих ограниченных контекстах (команды, запросы, события и саги).

На этом завершается этап моделирования в процессе предметно-ориентированного проектирования, и теперь у нас есть все необходимое, для того чтобы начать этап реализации.

В этой книге подробно рассматриваются четыре независимые реализации предметно-ориентированного проектирования с использованием платформы Enterprise Java как основы для разработки этих реализаций:

- монолитная версия с использованием платформы Java EE 8/Jakarta EE;
- версия с применением микросервисов на основе платформы Eclipse MicroProfile;
- версия с применением микросервисов на основе платформы Spring Boot;
- версия с применением микросервисов на основе непосредственного использования шаблона проектирования Command/Query Responsibility Segregation (CQRS)/Event Sourcing (ES) (разделение ответственности

команд и запросов/выбор источников событий[1]) в рабочей программной среде Axon Framework.

Платформа Enterprise Java предлагает обширную экосистему инструментальных средств, рабочих сред и методик, которые помогут нам реализовать концепции предметно-ориентированного проектирования, описанные в предыдущих главах.

В этой главе подробно рассматривается первая реализация предметно-ориентированного проектирования в приложении Cargo Tracker с использованием платформы Java EE 8 как основы для этой реализации. Приложение Cargo Tracker будет проектироваться как модульное монолитное приложение. При этом артефакты предметно-ориентированного проектирования будут отображаться в соответствующие элементы реализации, доступные на платформе Java EE 8.

Но для начала сделаем краткий обзор платформы Java EE.

Платформа Java EE

Платформа Java EE (Enterprise Edition) в течение почти 20 лет представляла собой фактический стандарт разработки приложений масштаба предприятия. Эта платформа предлагает набор спецификаций, охватывающих диапазон технических функциональных возможностей, необходимых предприятиям для создания легко масштабируемых, безопасных и надежных приложений стандартным способом.

Главная цель платформы Java EE – упростить деятельность разработчиков, позволяя им формировать собственные бизнес-функциональности, тогда как сама платформа выполняет всю самую трудную работу по обеспечению системных сервисов посредством реализации набора спецификаций с использованием сервера приложений (Application Server).

Поддерживаемая компанией Oracle, эта платформа получила всеобщее признание и широкое распространение. Ее также поддерживает крупное сообщество и около 15 сторонних поставщиков реализаций разнообразных спецификаций. Текущая версия платформы Java EE 8 с сервером приложений Oracle GlassFish v.5.0 используется в рассматриваемом в этой книге приложении.

Смена торговой марки на Jakarta EE и дальнейшее развитие

В 2017 году компания Oracle при поддержке компаний IBM и Red Hat решила передать исходный код платформы Java EE под управление некоммерческой координирующей организации Eclipse Foundation в рамках нового проекта EE4J (Eclipse Enterprise for Java). Целью этой передачи было создание более

[1] В русскоязычной литературе по предметно-ориентированному проектированию перевод термина Event Sourcing пока еще не является точно определенным, поэтому здесь предлагается вариант «выбор источников событий». — *Прим. перев.*

гибкого управляющего процесса с ускоренной периодичностью выпуска новых версий для полного соответствия стремительно развивающейся технологической среде в промышленности.

EE4J представляет собой проект самого высокого уровня в рамках Eclipse Foundation, в который постоянно передаются все исходные коды Java EE, эталонные реализации и комплекты тестов TCK (Technology Compatibility Kit). Платформа Jakarta EE – это проект, входящий в состав проекта верхнего уровня EE4J, позиционируемый как будущая платформа, которая полностью заменит ныне существующую платформу Java EE.

> **ПРИМЕЧАНИЕ** Если говорить кратко, то все новые версии технических спецификаций и спецификаций поддержки и сопровождения теперь будут создаваться в проекте Jakarta EE, а Java EE 8 должна стать самой последней версией этой платформы.

Jakarta EE уже продемонстрировала потрясающий импульс развития при поддержке многочисленных участников рабочих комиссий и нацеленность на модернизацию стека при сохранении поддержки ранее существующих обычных корпоративных приложений и в то же время с обеспечением поддержки новых облачных архитектур и архитектур на основе микросервисов.

Первая версия платформы Jakarta EE должна представлять собой точное воспроизведение платформы Java EE 8, при этом главное внимание уделено процессу передачи различных спецификаций между Oracle и Eclipse Foundation. Первая эталонная реализация под торговой маркой платформы Jakarta EE была опубликована как Eclipse GlassFish 5.1 и сертифицирована как полностью совместимая с платформой Java EE 8.

В этой главе все внимание сосредоточено на версии GlassFish, совместимой с платформой Java EE 8 и ее первым релизом на платформе Jakarta EE – Eclipse GlassFish 5.1. Цель этой главы – реализация концепций предметно-ориентированного проектирования в конкретном приложении Cargo Tracker, основанном на традиционной монолитной архитектуре.

Рассмотрим более подробно спецификации платформы.

Спецификации платформы Jakarta EE

Спецификации платформы Jakarta EE (основанной на платформе Java EE 8) имеют огромный объем и предназначены для предоставления стандартного набора функциональных возможностей, которые требуются промышленным предприятиям для создания программных приложений. Эти спецификации постоянно развиваются в многочисленных версиях, а функциональные возможности добавляются или обновляются либо в форме новых спецификаций, либо как изменяемые (сопровождаемые) версии спецификаций.

Спецификации группируются по двум профилям – полный профиль (Full Profile) и веб-профиль (Web Profile). Концепция профилей была введена для разделения на категории функциональных возможностей, требуемых для приложений. Для исключительно ориентированных на веб-приложения спецификация веб-профиля предоставляет требуемый набор функциональных воз-

можностей (Java Persistence API [JPA], Contexts and Dependency Injection [CDI], Java API for RESTful Web Services [JAX-RS]). Более сложные приложения могут потребовать, например, функции обмена сообщениями или поддержку интеграции с более старыми приложениями, в этом случае спецификация полного профиля предоставляет дополнительные функциональные возможности.

Для наших целей вполне подходит веб-профиль (Web Profile), который поможет реализовать приложение Cargo Tracker в монолитном архитектурном стиле, поэтому воспользуемся только этим набором спецификаций.

Набор спецификаций Web Profile для Jakarta EE (на основе платформы Java EE 8) показан на рис. 3.1 и разделен на группы по областям их применения. Официальный URL, на котором доступны все эти спецификации, – www.oracle.com/technetwork/java/javaee/tech/index.html.

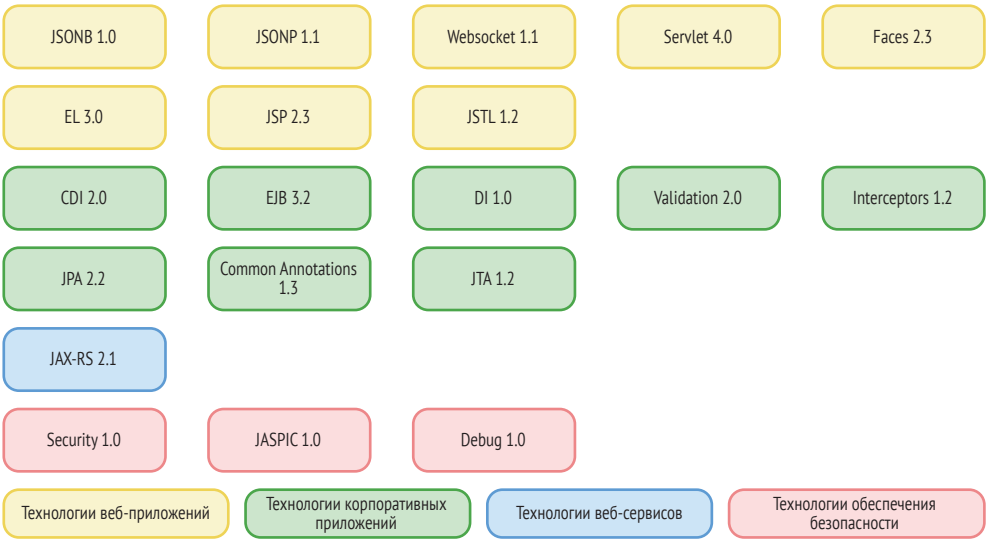

Рис. 3.1 ❖ Набор спецификаций Web Profile для платформы Jakarta EE (на основе платформы Java EE 8)

Технологии веб-приложений

Технологии веб-приложений представляют все спецификации платформы Jakarta EE, которые охватывают следующие области применения:

- функции обработки запросов и ответов по протоколу HTTP;
- компоненты языка HTML для создания тонких клиентских приложений на основе браузера;
- функции обработки данных в формате JSON.

Сервлет Java

По существу сервлеты (servlets) принимают запросы по протоколу HTTP(S), обрабатывают их и отправляют ответы клиенту, которым обычно является веб-браузер. Спецификация сервлета – один из наиболее важных документов,

существующих с версии Java EE 1.0 и служащих основной технологией для веб-приложения.

Множество рабочих программных веб-сред (например, JavaServer Faces [JSF], Spring Model View Controller [MVC]) используют сервлеты как основной набор инструментальных средств и абстрагируют их применение, т. е. сервлеты чрезвычайно редко используются напрямую при работе с программными веб-средами.

В Java EE 8 самой последней версией спецификации сервлетов является Servlet 4.0, в которой представлена главная функциональная возможность Servlet API – поддержка протокола HTTP/2. Обычные запросы HTTP ранее выполнялись как отдельные запросы/ответы, но с вводом протокола HTTP/2 вы можете выполнить один запрос, а сервер может сформировать несколько ответов одновременно. Таким образом, оптимизируются ресурсы и повышается качество обслуживания пользователя.

JavaServer Faces

JavaServer Faces обеспечивает основанную на компонентах методику создания веб-приложений. Вся обработка выполняется на стороне сервера, поэтому JavaServer Faces (JSF) реализует хорошо известный шаблон проектирования MVC с четким разделением функций. Представления (Views) главным образом основаны на технологии шаблонов (темплетов) Facelets JSF. Модели (Models) формируются с использованием JSF Backing Beans. Контроллер (Controller) создается поверх Servlet API.

JSF применяется весьма широко и постоянно выделяется среди наиболее популярных рабочих веб-сред, принимаемых корпоративными клиентами для создания собственных веб-приложений, благодаря основательно проработанным принципам проектирования и стабильности спецификации JSF. Существуют многочисленные реализации JSF, в том числе Oracle ADF Faces, PrimeFaces и BootsFaces.

Самая последняя версия спецификации – JavaServer Faces 2.3.

JavaServer Pages

JavaServer Pages (JSP) – самая первая технология представлений (view), которая была предложена при создании платформы Java EE. Страницы JSP преобразовываются в сервлеты во время выполнения, тем самым помогая создавать динамический веб-контент в веб-приложениях Java. В настоящее время страницы JSP уже не используются столь широко из-за преобладающей роли JSF как технологии пользовательского интерфейса (UI) для веб-приложений Java, поэтому спецификация не обновлялась в течение длительного времени.

Самая последняя версия спецификации – JavaServer Pages 2.3.

Expression Language

Спецификация Expression Language (EL) помогает обеспечить доступ к данным и их обработку. EL используется многими спецификациями, в том числе JSP, JSF и CDI. EL – достаточно мощный язык, поэтому широко применяется. Среди недавних усовершенствований следует особо выделить поддержку лямбда-выражений, введенных в Java 8.

Самая последняя версия спецификации – Expression Language 3.0.

JSP Standard Tag Library (JSTL)

Стандартная библиотека тегов JSP Standard Tag Library (JSTL) предлагает набор полезных тегов, которые можно использовать в страницах JSP. Эти теги позволяют выполнять такие задачи, как итерации, обработка условных выражений, команды SQL для доступа и обработки данных. После внедрения JSF спецификация не обновлялась, и JSTL сейчас используется редко.

Текущая версия спецификации – JSTL 1.2.

Java API для WebSocket

Эта спецификация обеспечивает возможность интеграции сокетов WebSocket в веб-приложения Java. В спецификации подробно описан API, который охватывает как сторону сервера, так и сторону клиента в реализациях сокетов WebSocket.

Эта спецификация была разработана и сопровождалась в среде Java EE 8, и ее последняя версия – 1.1.

Java API для связывания с форматом JSON

Это новая спецификация, введенная в Java EE 8, она подробно описывает API, предоставляющий уровень связывания для преобразования объектов Java в сообщения JSON, и наоборот.

Это первая версия спецификации, поэтому ее номер – 1.0.

Java API для обработки формата JSON

Эта спецификация предоставляет API, который можно использовать для доступа и обработки объектов JSON. Самая последняя версия спецификации в среде Java EE 8 была основным релизом с разнообразными усовершенствованиями, такими как JSON Pointer, JSON Patch, JSON Merge Patch и JSON Collectors.

Текущая версия спецификации – 1.1.

Технологии корпоративных приложений

Технологии корпоративных приложений (приложений масштаба промышленного предприятия) представляют все спецификации на платформе Jakarta EE, которые охватывают следующие области применения:

- создание корпоративных бизнес-компонентов;
- функциональные возможности Business Component Dependency Management/Injection;
- функциональные возможности валидации;
- управление транзакциями;
- функциональные возможности ORM (Object Relational Mapping).

Enterprise Java Beans (3.2)

Доступные с версии 1.0 на платформе Java EE, Enterprise Java Beans (EJB) предоставляют стандартный способ реализации бизнес-логики на стороне сервера для корпоративных приложений. EJB позволяют разработчику абстрагироваться от большой группы инфраструктурных комплексов задач (например,

обработка транзакций, управление жизненным циклом) и сосредоточиться исключительно на бизнес-логике. Это одна из наиболее часто применяемых спецификаций, но она очень сложна для восприятия и для практического использования. В спецификации были проработаны основные преобразования для ясного определения и решения проблем низкого уровня. В последней версии спецификации предлагается предельно простая и понятная модель программирования для создания бизнес-объектов.

Самая последняя версия спецификации – Enterprise Java Beans 3.2.

Contexts and Dependency Injection для Java (2.0)

Contexts and Dependency Injection (CDI – контексты и механизм инъекции зависимостей) были представлены в спецификации Java EE для создания отдельного компонента и управления его зависимостями через механизм инъекции. В этой спецификации вводилось ограничение зоны ответственности EJB периферийными инфраструктурными компонентами, тогда как основная бизнес-логика пишется в CDI Beans. В последних релизах платформы эти инфраструктурные компоненты теперь также могут быть написаны в CDI Beans. В настоящее время CDI становится основной частью технологии почти для всех остальных частей платформы, а EJB постепенно отодвигаются на второй план. Основной релиз этой спецификации был выпущен в Java EE 8 с поддержкой асинхронных событий, упорядочением шаблона наблюдателя (observer) и согласованием с потоками Java 8.

Одним из самых важных свойств CDI является расширение рабочей среды, позволяющее создавать функциональные возможности, которые пока еще не поддерживаются стандартным набором спецификаций. В список таких функциональных возможностей можно включить следующие:

- интеграцию с новыми брокерами сообщений (например, Kafka);
- интеграцию с нереляционными хранилищами данных (например, MongoDB, Cassandra);
- интеграцию с новой облачной инфраструктурой (например, AWS S3, Oracle Object Storage).

Широко известными расширениями CDI являются Apache DeltaSpike (https://deltaspike.apache.org/) и Arquillian (http://arquillian.org/).

Самая последняя версия этой спецификации – 2.0.

Валидация компонентов Bean

Эта спецификация представляет прикладной программный интерфейс Java API для реализации валидаций в приложениях. Основной релиз этой спецификации вышел в Java EE 8 с поддержкой новых типов валидации, с интеграцией с новым интерфейсом Java Time AI и т. д.

Самая последняя версия этой спецификации – 2.0.

Java Persistence API (JPA)

Эта спецификация представляет прикладной программный интерфейс Java API для реализации функций отображения ORM (Object Relational Mapping) между объектами Java Objects и реляционными хранилищами данных. Это одна из наиболее часто используемых спецификаций, она широко применя-

ется на практике и имеет многочисленные реализации, самой известной из которых является Hibernate.

Самая последняя версия этой спецификации – 2.2.

Java Transaction API (JTA)

Эта спецификация представляет прикладной программный интерфейс Java API для реализации программных транзакционных функций в приложениях. Этот API поддерживает распределенные транзакции между несколькими репозиториями. Это один из наиболее важных аспектов для монолитных приложений, для которых чрезвычайно важна высокая степень целостности и согласованности транзакций.

Самая последняя версия этой спецификации – 1.2.

Общие аннотации (Common Annotations)

Эта спецификация предоставляет набор аннотаций или маркеров, которые оказывают помощь контейнеру в выполнении общих задач (например, инъекции ресурсов (resource injections), управление жизненным циклом).

Самая последняя версия этой спецификации – 1.3.

Перехватчики (Interceptors)

Эта спецификация помогает разработчикам написать методы перехвата (interceptor) в связанных управляемых компонентах beans (EJB, CDI). Методы-перехватчики чаще всего используются для централизованного выполнения обобщенных, «пронизывающих» всю структуру приложения задач, таких как аудит и ведение журналов.

Самая последняя версия этой спецификации – 1.2.

Веб-сервисы в Jakarta EE

Технологии веб-сервисов представлены на платформе Jakarta EE всеми спецификациями, которые охватывают область создания REST-сервисов масштаба промышленного предприятия. В настоящее время существует один основной API.

Java API for RESTful Web Services (JAX-RS)

Эта спецификация предоставляет разработчикам стандартный интерфейс Java API для реализации RESTful веб-сервисов. Это еще одна часто применяемая спецификация, в основном релизе самой последней версии которой обеспечена поддержка реактивных клиентов (Reactive Clients) и событий на стороне сервера (Server-Side Events).

Самая последняя версия этой спецификации – 2.1.

Технологии обеспечения безопасности

Технологии обеспечения безопасности представлены на платформе Jakarta EE всеми спецификациями, которые охватывают область обеспечения защиты бизнес-компонентов масштаба промышленного предприятия.

Java EE Security API (1.0)

Это новая спецификация, введенная в Java EE 8, которая представляет стандартный интерфейс Java API для реализаций средств защиты, основанных на управлении пользователем. Были введены новые API для управления аутентификацией, для идентификации взаимодействий между хранилищами данных, а также для реализаций контекста защиты (безопасности) (корректирующее восстановление пользовательской информации).

Итоговый обзор спецификаций Jakarta EE

На этом завершается краткий общий обзор спецификаций платформы Jakarta EE, основанной на платформе Java EE 8. Как мы убедились, эти спецификации являются полными и исчерпывающими и представляют практически каждую функциональную возможность, необходимую для создания приложений масштаба промышленного предприятия. Кроме того, платформа предоставляет пункты расширения для тех случаев, когда возникает необходимость в реализации каких-либо специфических потребностей промышленного предприятия.

Здесь самым важным является тот факт, что это стандартные спецификации, сформированные многочисленными участниками в соответствии с существующими стандартами. Это предоставляет промышленным предприятиям чрезвычайную гибкость при выборе платформы развертывания.

С новой структурой управления, принятой в организации Eclipse Foundation, платформа готовится к созданию нового поколения приложений масштаба промышленного предприятия (корпоративных приложений).

Cargo Tracker как модульное монолитное приложение

Монолитная архитектура ПО являлась основой для программных проектов масштаба промышленного предприятия в течение весьма длительного времени.

Для монолитной архитектуры характерны следующие основные особенности:

- строгая транзакционная целостность и согласованность;
- более удобное сопровождение;
- централизованное управление данными;
- разделение уровней ответственности.

Недавнее внедрение технологии микросервисов привело к постоянно возрастающему давлению и вытеснению монолитных архитектур. Архитектура микросервисов предоставляет группам разработчиков высокую степень независимости на этапах разработки, тестирования и развертывания приложений. Тем не менее требуется особое внимание и осторожность, когда вы начинаете демонтаж монолитного приложения и переходите на архитектуру микросервисов. Микросервисы – это, по существу, распределенная система, которая в свою очередь требует огромных вложений в автоматизацию, мониторинг

и регулирование целостности и согласованности. Монолитные программы обладают значительной ценностью для сложных бизнес-приложений.

Но подход к использованию монолитных архитектур изменился при заимствовании концепций из технологии микросервисов, особенно в области структурирования монолитных приложений. Именно здесь предметно-ориентированное проектирование играет самую важную роль. Мы уже видели, что ограниченные контексты помогают четко определить бизнес-функции конкретных предметных областей (доменов) и разделить их на независимые области решений (solution areas). Структурирование этих ограниченных контекстов как отдельных модулей в монолитном приложении и использование событий предметной области (домена) для обмена данными между ними помогает обеспечить меньшую связность, т. е. «истинную модульность» (true modularity), которую также обозначают термином «модульные монолиты» (modular monoliths).

Преимущество выбора варианта истинной модульности или модульных монолитов с использованием предметно-ориентированного проектирования состоит в том, что при сохранении всех положительных свойств монолитной архитектуры этот вариант обеспечивает создание уровня независимости с перспективой дальнейшего перехода на микросервисы, если это будет необходимо.

В предыдущих главах были подробно описаны бизнес-функции/поддомены для рассматриваемого в этой книге приложения Cargo Tracker решения для них с помощью ограниченных контекстов. В этой главе мы займемся структурированием приложения Cargo Tracker как модульной монолитной программы с каждым ограниченным контекстом, смоделированным как отдельный модуль.

На рис. 3.2 показано отображение ограниченных контекстов в соответствующие модули монолитного приложения Cargo Tracker.

Располагая набором спецификаций, описанных выше, и четко поставленной задачей для приложения Cargo Tracker, для которого здесь принята архитектура модульной монолитной программы на основе предметно-ориентированного проектирования, продолжим реализацию этого приложения на платформе Java EE.

Ограниченные контексты с использованием платформы Jakarta EE

Ограниченный контекст – это исходный пункт этапа решения для нашей реализации предметно-ориентированного проектирования для монолитного приложения Cargo Tracker. Каждый ограниченный контекст должен быть структурирован как модуль в монолитном приложении, т. е. как независимо развертываемый артефакт этого приложения.

Основным артефактом развертывания монолитного приложения Cargo Tracker будет стандартный файл в формате WAR (Web Archive), развертываемый на сервере приложений (Eclipse GlassFish). Как уже отмечалось выше, сер-

вер приложений предоставляет реализацию для специализированной версии спецификации Jakarta EE (в данном случае Java EE 8). Артефакт развертывания каждого ограниченного контекста будет стандартным файлом в формате JAR (Java Archive), который будет связан с файлом WAR (размещен в этом файле).

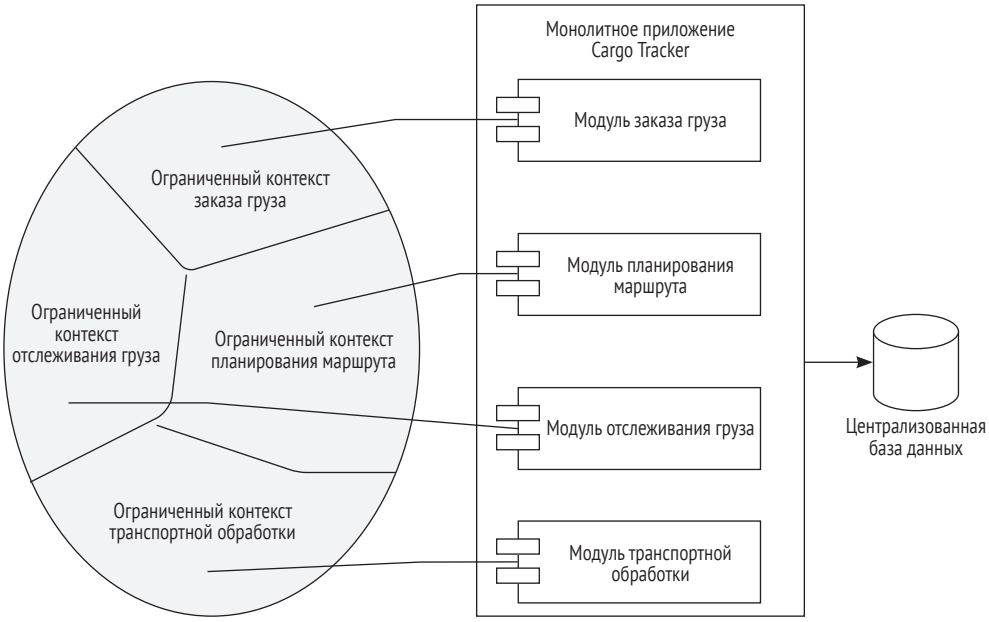

Рис. 3.2 ❖ Ограниченные контексты как модули в монолитном приложении Cargo Tracker с централизованной базой данных

Этот WAR-файл должен содержать набор JAR-файлов. Каждый JAR-файл представляет модуль/ограниченный контекст. Архитектура развертывания показана на рис. 3.3.

Реализация ограниченных контекстов подразумевает логическое группирование ранее определенных артефактов предметно-ориентированного проектирования в единый развертываемый артефакт. При логическом группировании предполагается определение структуры пакета, где размещаются различные артефакты предметно-ориентированного проектирования для получения обобщенного решения для ограниченного контекста. Таким образом, мы не используем какую-либо конкретную спецификацию Java EE для реализации ограниченного контекста. Вместо этого четко определяется структура пакета для ранее идентифицированных артефактов предметно-ориентированного проектирования в ограниченном контексте.

Структура пакета должна зеркально отображать гексагональную архитектуру, описанную в главе 2 (рис. 2.16).

Структура пакета для любого используемого в рассматриваемом здесь примере приложения ограниченного контекста показана на рис. 3.3.

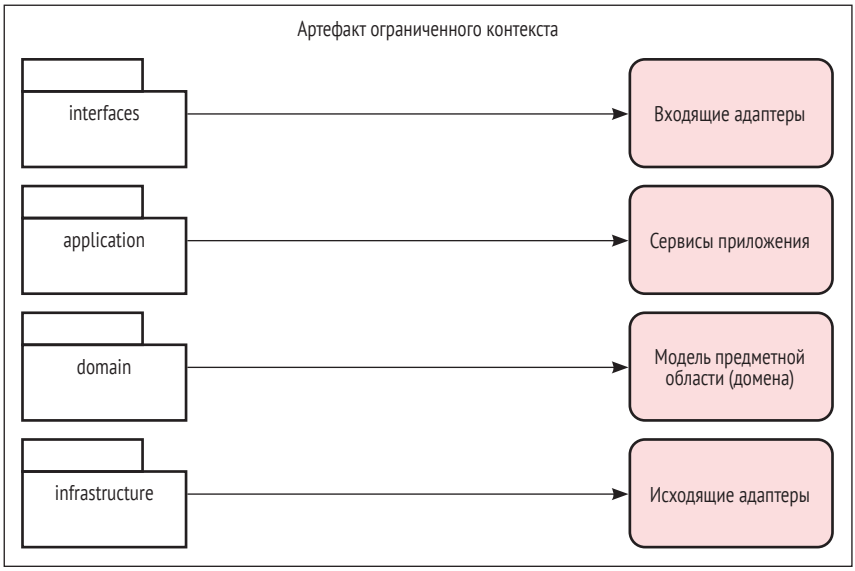

Рис. 3.3 ❖ Структура пакета для ограниченных контекстов

Рассмотрим подробнее структуру пакетов приложения.

Пакет interfaces

Рассматриваемый пакет содержит все возможные входящие сервисы, предоставляемые ограниченным контекстом и классифицированные по протоколам.

При этом преследуются две основные цели:

- согласование протоколов со стороны модели предметной области (домена) (например, REST API, веб-API, WebSocket, FTP);
- представление адаптеров для данных (например, представления браузеров, мобильные представления).

Ограниченный контекст заказа груза предоставляет несколько типов сервисов. Один из примеров – веб-API для собственного пользовательского интерфейса, встроенного в приложение Cargo Tracker для заказа груза или изменения условий заказа, а также для предъявления списка грузов клиенту. Ограниченный контекст транспортной обработки груза предоставляет RESTful API для любого типа операций обработки груза, которые используются мобильным приложением транспортной обработки груза. Все эти сервисы должны быть частью пакета interfaces.

Структура пакета interfaces показана на рис. 3.4.

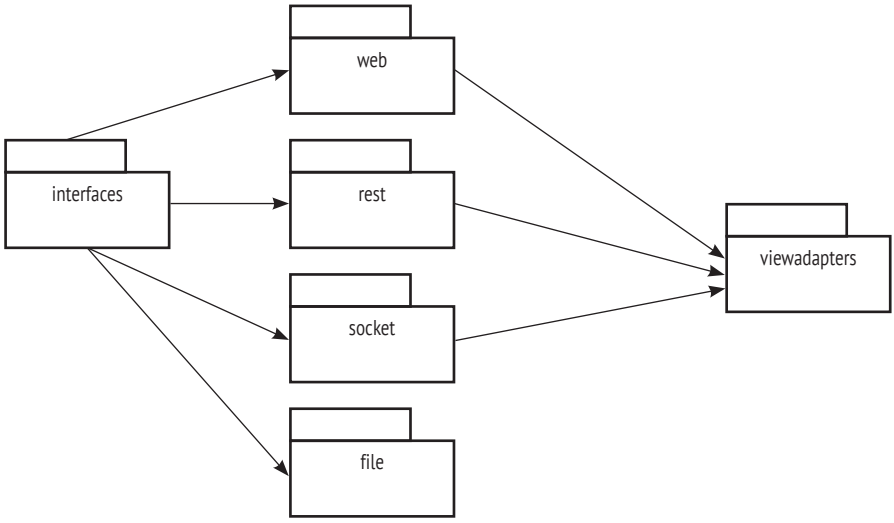

Рис. 3.4 ❖ Структура пакета interfaces

Пакет application

Этот пакет содержит сервисы приложения, которые потребуются модели предметной области (домена) ограниченного контекста.

Классы сервисов приложения предназначены для нескольких целей:

- выполнение роли портов для входящих интерфейсов и исходящих репозиториев;
- участие в командах, запросах, событиях и сагах;
- инициализация, управление и завершение транзакций;
- выполнение централизованных функций (например, ведение журналов, обеспечение безопасности, определение метрик) для модели предметной области (домена) более низкого уровня;
- преобразование объекта передачи данных;
- исходящие вызовы, направленные в другие ограниченные контексты.

Структура пакета application показана на рис. 3.5.

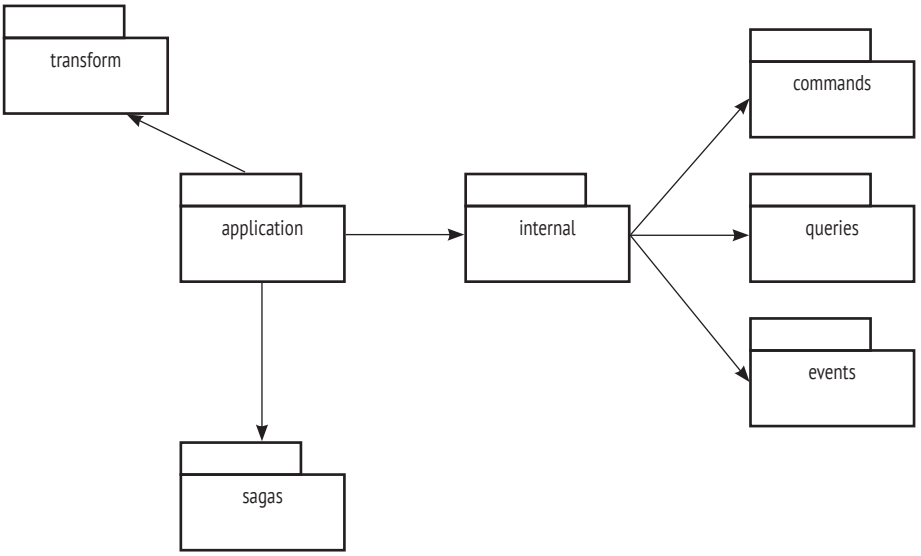

Рис. 3.5 ❖ Структура пакета для сервисов приложения

Пакет domain

Этот пакет содержит модель предметной области (домена) ограниченного контекста.

Основные классы для ограниченного контекста:

- агрегаты;
- сущности;
- объекты-значения;
- правила предметной области (домена).

Структура пакета domain показана на рис. 3.6.

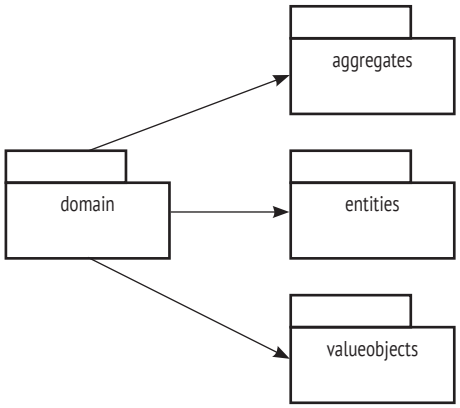

Рис. 3.6 ❖ Структура пакета для модели предметной области

Пакет infrastructure

Этот пакет содержит компоненты инфраструктуры, необходимые модели предметной области (домена) ограниченного контекста для обмена данными со всеми внешними репозиториями (например, с реляционными базами данных, с базами данных типа NoSQL, с очередями сообщений, с инфраструктурой событий).

Структура пакета infrastructure показана на рис. 3.7.

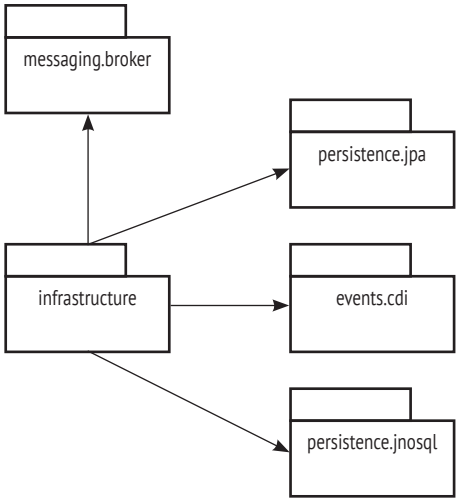

Рис. 3.7 ❖ Структура пакета для компонентов инфраструктуры

Совместно используемые ядра

Иногда может потребоваться совместное использование модели предметной области (домена) несколькими ограниченными контекстами. Совместно используемые ядра (shared kernels) в предметно-ориентированном проектировании предлагают надежный механизм совместного использования моделей предметной области (домена), позволяющий сократить объем повторяющегося кода. Совместно используемые ядра проще реализовать в монолитном приложении, чем в приложении на основе микросервисов, которые поддерживают гораздо более высокий уровень независимости.

При реализации механизма совместно используемых ядер возникают специфические трудности, так как несколько групп разработчиков должно договориться о том, какие именно аспекты модели предметной области (домена) будут совместно использоваться несколькими ограниченными контекстами.

Для рассматриваемого здесь примера в монолитном приложении Cargo Tracker все события (пакет – events.cdi), генерируемые различными ограниченными контекстами, будут сохраняться в совместно используемом ядре, как показано на рис. 3.8.

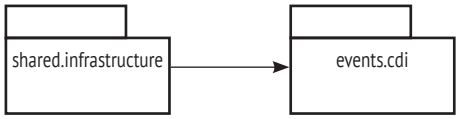

Рис. 3.8 ❖ Совместно используемая инфраструктура, содержащая все события CDI

Теперь все ограниченные контексты аккуратно сгруппированы по модулям в структуре пакетов с отчетливо разделенными функциями.

Реализация модели предметной области (домена) с использованием Jakarta EE

В рассматриваемом примере модель предметной области (домена) является центральной функцией ограниченного контекста и, как уже отмечалось выше, имеет связанный с ней набор артефактов. Реализация этих артефактов выполняется с помощью инструментальных средств, предоставляемых платформой Java EE.

Ниже перечислены артефакты модели предметной области (домена), которые необходимо реализовать:

- агрегаты;
- сущности;
- объекты-значения.

Рассмотрим более подробно каждый из этих артефактов и определим соответствующие инструментальные средства, которые платформа Java EE предоставляет для их реализации.

Агрегаты

Агрегаты – основные элементы модели предметной области (домена). Напомним, что ранее были определены четыре агрегата в каждом из ограниченных контекстов, как показано на рис. 3.9.

Реализация агрегатов охватывает следующие аспекты:

- реализацию класса агрегата;
- полноту предметной области (домена) (бизнес-атрибуты, бизнес-методы);
- формирование состояния;
- инерционность состояния;
- ссылки между агрегатами;
- события.

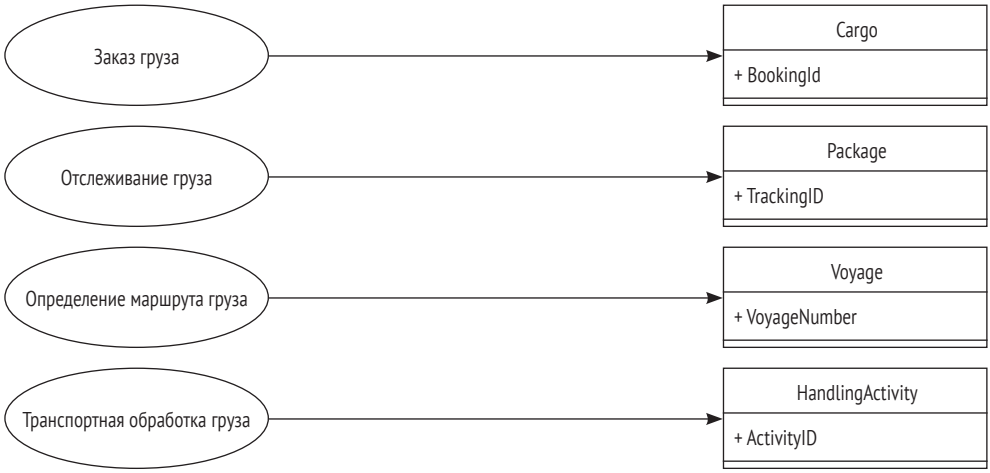

Рис. 3.9 ❖ Агрегаты в ограниченных контекстах

Реализация класса агрегата

Для реализации корневого агрегата как основное инструментальное средство будет использоваться JPA (Java Persistence API) из рабочей среды Java EE. Каждый класс корневого агрегата реализуется как объект JPA. В JPA не предусмотрено специальной аннотации для конкретного класса как корневого агрегата, поэтому используется стандартная аннотация `@Entity`, предоставляемая JPA.

В листинге 3.1 показана реализация корневого агрегата Cargo Root.

Листинг 3.1 ❖ Агрегат Cargo Root

```
package com.practicalddd.cargotracker.booking.domain.model.aggregate;

import javax.persistence.Entity;
@Entity      // Аннотация, предоставляемая JPA.
public class Cargo implements Serializable {
@Id
@GeneratedValue
private Long id;     // Ключ-идентификатор.
@Embedded      // Для сохранения полноты предметной области (домена) используется
               // встроенный класс вместо прямой реализации на языке Java.
private BookingId bookingId     // Глобальный уникальный идентификатор (Booking Id)
                                // агрегата Cargo Root.
}
```

В листинге 3.2 показана реализация идентификатора агрегата `BookingId`.

Листинг 3.2 ❖ Идентификатор агрегата BookingId

```
@Embeddable
public class BookingId implements Serializable {
@Column( name="booking_id", unique=true,updateable=false )
private String id;
public BookingId() {
}
```

```
public BookingId( String id ) {
  this.id = id;
}

public String getBookingId() {
  return id;
}
```

Для реализации идентификатора агрегата был выбран технический ключ-идентификатор (первичный ключ – Primary Key) и соответствующий бизнес-ключ (уникальный ключ – Unique Key). Бизнес-ключ ясно выражает бизнес-предназначение идентификатора агрегата, т. е. это идентификатор нового заказанного груза и это ключ, который предъявляется пользователям модели предметной области (домена). С другой стороны, технический ключ является исключительно внутренним представлением идентификатора агрегата и полезен для некоторых вариантов использования, таких как ссылки между агрегатами.

JPA предоставляет аннотацию `@Id` для обозначения первичного ключа корневого агрегата.

Сравнение агрегата полноценного домена и неполноценных агрегатов

Основной предпосылкой предметно-ориентированного проектирования является полнота (полноценность) предметной области (домена), выраженная и централизованная в модели предметной области (домена). Агрегаты в рассматриваемом здесь примере формируют основу модели предметной области (домена).

Такой агрегат должен соответствовать полноценной предметной области (домена) и выражать предназначение ограниченного контекста, используя точно сформулированные бизнес-концепции.

Агрегат также может в конечном итоге стать неполноценным (anemic), т. е. объектом, в котором имеются только методы считывания (getters) и установки (setters) значений. Такой агрегат считается анти-шаблоном в области предметно-ориентированного проектирования.

Ниже приводятся краткие характеристики неполноценных агрегатов:

- неполноценные агрегаты не выражают цель или предназначение конкретной предметной области (домена);
- такой шаблон предназначен только для хранения атрибутов и наиболее полезен в объектах преобразования передаваемых данных, но не в качестве основных бизнес-объектов;
- применение неполноценных агрегатов в конечном итоге приводит к утечке логики предметной области (домена) в окружающие сервисы, в результате чего создается так называемое загрязнение предназначения этих окружающих сервисов;
- неполноценные агрегаты через некоторое время становятся причиной того, что сопровождение кода становится чрезвычайно затрудненным или даже невозможным.

Следует избегать использования неполноценных агрегатов во всех случаях, когда это возможно, и ограничивать их преднамеренное применение особыми случаями, например, в качестве чистых объектов данных.

С другой стороны, агрегаты полноценной предметной области (домена) в соответствии со своим наименованием являются полноценными (rich). Полноценные агрегаты явно выражают предназначение поддомена, который они представляют через бизнес-атрибуты и бизнес-методы. В следующих подразделах они будут рассматриваться более подробно.

Охват бизнес-атрибутов

Корневой агрегат должен охватывать (покрывать) все бизнес-атрибуты, необходимые для функционирования конкретного ограниченного контекста. Эти атрибуты должны моделироваться в терминах бизнеса, а не в технических терминах.

Приступим к подробному рассмотрению примера корневого агрегата Cargo.

Корневой агрегат (объект) Cargo (Груз) содержит следующие атрибуты:

- исходную локацию;
- количество (объем, массу) заказываемого груза;
- спецификацию маршрута доставки (исходную локацию/конечный пункт/предельный срок доставки в конечный пункт);
- план маршрута доставки;
- ход процесса доставки груза.

Класс корневого агрегата Cargo содержит эти атрибуты как отдельные классы в основном классе агрегата.

В листинге 3.3 показаны аннотации атрибутов в корневом агрегате Cargo.

Листинг 3.3 ❖ Корневой агрегат Cargo – охват бизнес-атрибутов

```
@ManyToOne // Аннотация, предоставляемая JPA.
private Location origin;
@Embedded  // Аннотация, предоставляемая JPA.
private CargoBookingAmount bookingAmount;
@Embedded  // Аннотация, предоставляемая JPA.
private RouteSpecification routeSpecification;
@Embedded  // Аннотация, предоставляемая JPA.
private Itinerary itinerary;
@Embedded  // Аннотация, предоставляемая JPA.
private Delivery delivery;
```

Следует отметить использование терминов бизнеса для обозначения этих зависимых классов, которые ясно выражают предназначение корневого агрегата Cargo.

Java Persistence API (JPA) предоставляет набор структурных (например, Embedded/Embeddable) и реляционных (например, ManyToOne) аннотаций, которые помогают определить класс корневого агрегата исключительно в бизнес-концепциях.

Связанные классы моделируются либо как объекты-сущности, либо как объекты-значения. Эти концепции будут описаны более подробно немного позже, а сейчас для ясности отметим, что объекты-сущности в ограниченном контексте обладают собственной идентичностью, но всегда существуют только в корневом агрегате, т. е. не могут существовать независимо. Кроме того, объекты-сущности никогда не изменяются на протяжении полного жизненного цикла своего агрегата. Объекты-значения не обладают собственной сущностью и без затруднений могут быть изменены в любом экземпляре агрегата.

Охват бизнес-методов

Еще одним важным аспектом агрегатов является выражение логики предметной области (домена) с помощью бизнес-методов. Этот аспект способствует полноценности предметной области (домена), что наиболее важно при предметно-ориентированном проектировании.

Агрегаты обязаны содержать логику предметной области (домена), которая требуется для функционирования конкретного поддомена. Например, при запросе для агрегата Cargo операции загрузки этот агрегат должен создать производный процесс хода доставки груза и предъявить его пользователю-потребителю. Это должно быть сделано с помощью методов предметной области (домена) внутри агрегата в отличие от реализации на уровнях поддержки.

Реализация бизнес-методов выполняется в форме простых методов внутри агрегата, которые работают с текущим состоянием этого агрегата. В листинге 3.4 демонстрируется эта концепция на примере двух бизнес-методов. Обратите внимание на то, что за реализацию логики предметной области (домена) отвечает именно агрегат, а не уровни поддержки.

Листинг 3.4 ❖ Корневой агрегат Cargo – бизнес-методы

```
public class Cargo{
  public void deriveDeliveryProgress() {
    // Здесь размещается код реализации метода.
  }
  public void assignToRoute(Itinerary itinerary) {
    // Здесь размещается код реализации метода.
  }
}
```

> **ПРИМЕЧАНИЕ** Полную реализацию методов см. в репозитории исходного кода для данной главы.

Создание состояния агрегата

Операция создания состояния агрегата может выполняться для нового агрегата или при загрузке существующего агрегата.

Создание нового агрегата выполняется так же просто, как использование конструкторов класса для объекта JPA. В листинге 3.5 показан конструктор для создания нового экземпляра класса корневого агрегата Cargo.

Листинг 3.5 ❖ Создание корневого агрегата Cargo

```
public Cargo( BookingId bookingId, RouteSpecification routeSpecification ) {
  this.bookingId = bookingId;
  this.origin = routeSpecification.getOrigin();
  this.routeSpecification = routeSpecification;
}
```

Другой механизм создания нового агрегата – использование шаблона проектирования Factory (фабрика), т. е. применение статических фабрик, которые возвращают новый агрегат.

В ограниченном контексте транспортной обработки груза Handling создается корневой агрегат операции транспортной обработки груза Handling Activi-

ty, зависящий от типа выполняемого действия. Определенные типы операций транспортной обработки груза не требуют связи с рейсом. Когда заказчик окончательно подтверждает прибытие и прием груза, для соответствующей операции обработки груза не требуется связь с рейсом. Но когда груз разгружается в порту, связанная с этим действием операция обработки груза выполняется с учетом рейса. Таким образом, здесь рекомендуется методика с использованием фабрики для создания различных типов агрегатов операций транспортной обработки груза Handling Activity.

В листинге 3.6 показана фабрика, которая создает экземпляры агрегатов типа Handling Activity. Класс фабрики реализован с использованием компонента CDI Bean, тогда как экземпляр агрегата создается с помощью обычного конструктора.

Листинг 3.6 ❖ Корневой агрегат Handling Activity

```
package com.practicalddd.cargotracker.handling.domain.model.aggregate;

@ApplicationScoped   // Область видимости CDI фабрики (Область видимости приложения
                     // определяется единственным экземпляром на уровне приложения).
public class HandlingActivityFactory implements Serializable {
  public HandlingActivity createHandlingActivity( Date registrationTime,
           Date completionTime, BookingId bookingId, VoyageNumber voyageNumber,
           UnLocode unlocode, HandlingActivity.Type type ) {
    if( voyage == null ) {
      return new HandlingActivity( cargo, completionTime, registrationTime,
                                   type, location );
    } else {
       return new HandlingActivity( cargo, completionTime, registrationTime,
                                    type, location, voyage );
    }
  }
}
```

Загрузку существующего агрегата, т. е. восстановление состояния агрегата можно выполнить двумя способами:

- домен из источника (Domain Sourced), в котором создается состояние агрегата с помощью загрузки текущего состояния конкретного агрегата непосредственно из хранилища данных;
- событие из источника (Event Sourced), в котором создается состояние агрегата с помощью загрузки пустого агрегата и воспроизведения всех событий, которые ранее произошли в этом конкретном агрегате.

Для рассматриваемого здесь примера реализации монолитного приложения будет использоваться агрегат с восстановлением состояния из источника (state-sourced aggregate).

Агрегат с восстановлением состояния из источника загружается с использованием класса репозитория инфраструктурных данных, который принимает первичный идентификатор (ключ) агрегата и загружает полную иерархию объекта для этого агрегата, включая все связанные с ним сущности и объекты-значения, из хранилища данных (например, из реляционной базы данных или из базы данных типа NoSQL).

Загрузка агрегатов с восстановлением состояния из источника в общем случае выполняется в сервисах приложения (см. раздел «Сервисы приложения» ниже).

Загрузка агрегата с восстановлением состояния из источника Cargo показана в листинге 3.7. Обычно этот код размещается в сервисах приложения.

Листинг 3.7 ❖ Корневой агрегат Cargo – загрузка состояния из репозитория

```
Cargo cargo = cargoRepository.find(bookingId);
```

В этом фрагменте кода используется класс инфраструктуры `CargoRepository`, который принимает идентификатор заказа груза (Cargo Booking ID) и загружает иерархию объекта Cargo, которая включает количество заказанного груза (Booking Amount), спецификацию маршрута доставки (Route Specification), план маршрута доставки (Itinerary) и ход процесса доставки (Delivery progress). Как часть общей реализации мы будем использовать конкретную реализацию JPA (класс `JPACargoRepository`), которая загружает агрегат Cargo из реляционной базы данных.

Подведем промежуточные итоги:

- новые агрегаты могут быть созданы с использованием обычных конструкторов или с применением статических фабрик (шаблон Factory);
- существующие агрегаты создаются с использованием домена из источника (Domain Sourced), т. е. выполняется загрузка состояния иерархии объектов агрегата непосредственно из хранилища данных с помощью классов репозитория.

Персистентность состояния агрегата

Операция персистентности (persistence) агрегата должна воздействовать только на состояние этого конкретного агрегата. Вообще говоря, агрегат не является персистентным (устойчивым) сам по себе, а полагается на репозитории для выполнения этих операций. В рассматриваемом здесь случае это репозитории JPA. Если для нескольких агрегатов требуется персистентность, то они должны быть одним из классов сервисов приложения.

Ссылки между агрегатами

Ссылки между агрегатами (inter-aggregate references) – это отношения, которые существуют между агрегатами и пересекают границы ограниченных контекстов. В примере монолитного приложения они реализованы как ассоциации (отношения), предоставляемые JPA.

В качестве примера в корневом агрегате HandlingActivity ограниченного контекста Handling (транспортная обработка груза) здесь приводится отношение многие к одному (many-to-one) с агрегатом Cargo через идентификатор заказа груза (Booking ID) как столбец соединения (join column; в реляционной базе данных).

Листинг 3.8 представляет это отношение.

Листинг 3.8 ❖ Отношения корневого агрегата

```
public class HandlingActivity implements Serializable {
  @ManyToOne
  @JoinColumn(name = "cargo_ id")
```

```
  @NotNull
  private Cargo cargo;   // Ссылка на другой агрегат реализована через отношение.
}
```

После этого при любой загрузке агрегата `HandlingActivity` из хранилища данных также загружается соответствующая ссылка на агрегат `Cargo`, реализованная через описанное выше отношение.

Возможны и другие способы проектирования ссылок на агрегаты, например можно просто воспользоваться идентификатором первичного ключа для агрегата Cargo, т. е. идентификатором заказа груза (Booking ID), и извлекать подробности о заказанном грузе с помощью вызовов сервисов. В другом варианте можно сохранить подмножество данных о заказанном грузе, требуемое в ограниченном контексте транспортной обработки груза Handling, и получать оповещения об изменениях в агрегате Cargo через события, генерируемые ограниченным контекстом заказа груза Booking, который является владельцем агрегата Cargo.

Выбор варианта реализации отношений между агрегатами всегда является предметом дискуссий. С точки зрения приверженца безупречно правильного применения предметно-ориентированного проектирования, следует тщательно избегать отношений между агрегатами, так как это свидетельствует об утечке данных (вследствие слабой изолированности), поэтому необходимо пересмотреть границы ограниченных контекстов. Но иногда необходимо принять более практичный подход с учетом потребностей приложения (например, обеспечение целостности транзакций) и функциональных возможностей поддерживающей платформы (например, инфраструктуры событий).

В примерах реализации на основе микросервисов мы приняли пуристский подход, но в монолитном приложении Cargo Tracker ссылки между агрегатами реализованы через отношения (ассоциации) JPA.

События агрегатов

При самом правильном подходе к предметно-ориентированному проектированию события предметной области (домена) всегда должны публиковаться агрегатами. Если какое-либо событие публикуется любой другой частью приложения (например, классом сервисов приложения), то оно считается техническим событием, а не событием бизнес-домена. Несмотря на неоднозначность и даже спорность такого определения, оно логически вытекает из того факта, что только агрегат оповещает о произошедшем изменении состояния.

Платформа Java EE не предоставляет какой-либо прямой функциональной возможности публикации события предметной области (домена) из уровня агрегата, созданного на основе JPA, поэтому соответствующая часть реализации перемещается на уровень сервисов приложения. В последующих главах будут более подробно рассматриваться функциональные возможности, предоставляемые инструментальным комплектом более низкого уровня, который обеспечивает поддержку публикации событий предметной области (домена) из уровня агрегата (например, как часть рабочей программной среды Spring Framework, проект Spring Data Commons предоставляет аннотацию `@DomainEvents`, которую можно добавить в агрегат JPA). Теоретически можно воспользоваться классами `EntityListener` для прослушивания событий жизненного

цикла агрегата более низкого уровня, но при таком подходе представлены изменения в данных сущности (Entity), а не сами бизнес-события.

Обобщенная схема рассматриваемой здесь реализации агрегата с использованием платформы Java EE показана на рис. 3.10.

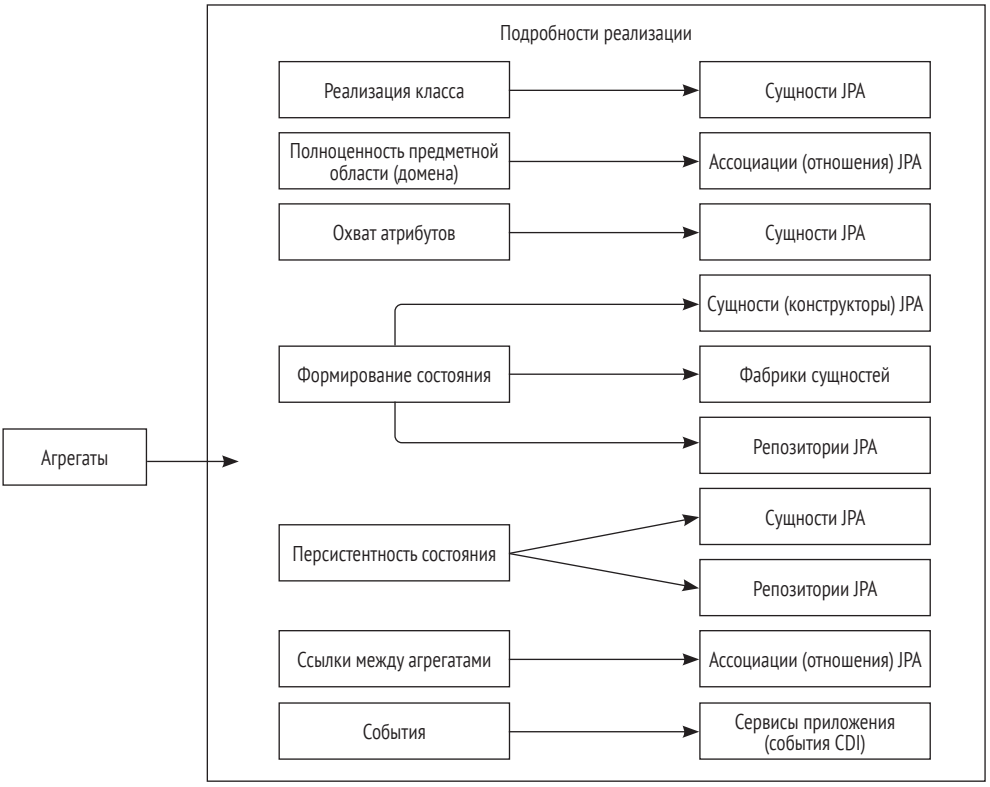

Рис. 3.10 ❖ Общая схема реализации агрегата

Сущности

Сущности (entities) в ограниченном контексте обладают собственной идентичностью, но всегда существуют только в корневом агрегате, т. е. не могут существовать независимо. Объект-сущность никогда не изменяется на протяжении полного жизненного цикла агрегата.

В примере, рассматриваемом в главе 2, в ограниченном контексте заказа груза Booking существует один объект-сущность – исходная локация (пункт отправки) груза (Origin Location). Пункт отправки груза никогда не изменяется на протяжении всего жизненного цикла этого груза, следовательно, это вполне подходящий кандидат на моделирование в качестве объекта-сущности.

Реализация объектов-сущностей охватывает следующие аспекты:

- реализацию класса сущности;
- определение отношений сущность–агрегат;
- формирование состояния сущности;
- персистентность состояния сущности.

Реализация класса сущности

Классы сущностей реализуются отдельно как сущности JPA с использованием стандартной аннотации `@Entity`, предоставляемой JPA.

Класс сущности `Location`, содержащий сгенерированный первичный ключ, код локации ООН (United Nations – UN) и описание, показан в листинге 3.9.

Листинг 3.9 ❖ Класс сущности Location

```
package com.practicalddd.cargotracker.booking.domain.model.entities;

import com.practicalddd.cargotracker.booking.domain.model.entities.UnLocode;

@Entity
public class Location implements Serializable {
  @Id
  @GeneratedValue
  private Long id;
  @Embedded
  private UnLocode unLocode;
  @NotNull
  private String name;
}
```

Идентификаторы сущностей используют ту же концепцию, что и идентификаторы агрегатов, т. е. технический ключ-идентификатор и бизнес-ключ.

Определение отношений сущность-агрегат

Классы сущностей имеют жестко определенные связи со своими корневыми агрегатами, т. е. они не могут существовать без корневого агрегата. Моделирование связи с корневым агрегатом выполняется с использованием стандартных аннотаций отношений JPA.

В корневом агрегате Cargo класс сущности Location используется для обозначения пункта отправки груза. Пункт отправки в ограниченном контексте заказа груза Booking не может существовать без конкретного действительно заказанного груза.

В листинге 3.10 показано отношение между классом сущности Location и корневым агрегатом Cargo.

Листинг 3.10 ❖ Определение отношения с корневым агрегатом Cargo

```
public class Cargo implements Serializable {
  @ManyToOne
  @JoinColumn( name = "origin_id", updatable = false )
  // Сущность Location не отвечает за обновление корневого агрегата, то есть Cargo
  private Location origin;
}
```

Формирование и персистентность состояния сущности

Формирование и обеспечение персистентности сущностей всегда осуществляется только совместно с соответствующим корневым агрегатом, когда эти операции выполняются в корневом агрегате.

Пункт отправки груза всегда формируется при создании агрегата Cargo. То же самое относится и к персистентности: когда обеспечена персистентность

нового заказа груза, вместе с этим обеспечивается и персистентность пункта отправки этого груза.

Обобщенная схема рассматриваемой здесь реализации сущности с использованием платформы Java EE показана на рис. 3.11.

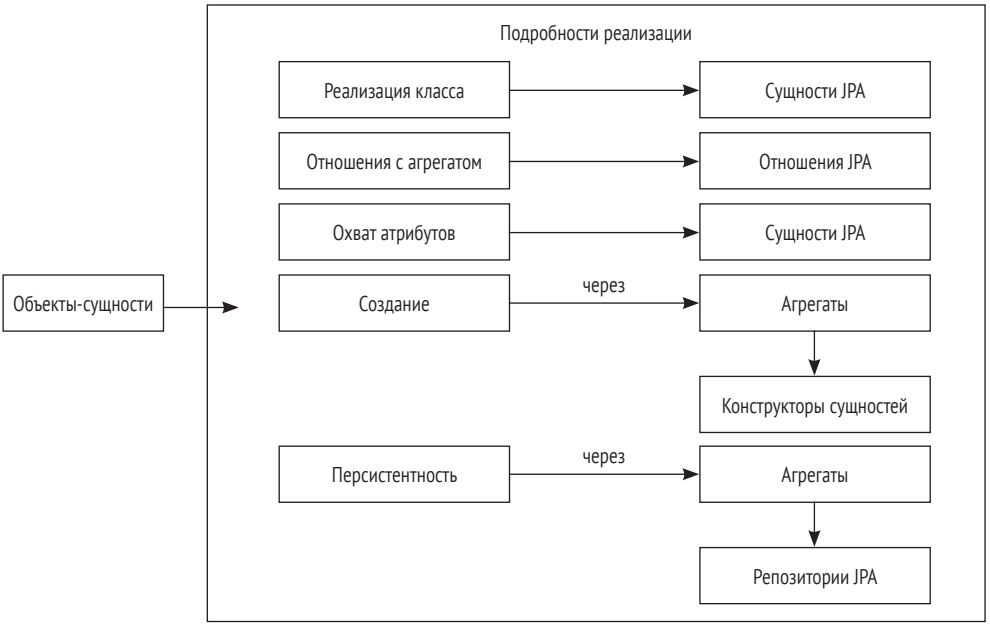

Рис. 3.11 ❖ Общая схема реализации сущности

Объекты-значения

Объекты-значения существуют в области видимости агрегата ограниченного контекста. Объекты-значения не обладают собственной идентичностью и могут быть заменены в любом экземпляре агрегата.

Снова обратимся к примеру из главы 2, где в ограниченном контексте заказа груза Booking имеется несколько объектов-значений, являющихся частью корневого агрегата Cargo:

- спецификация маршрута доставки груза;
- план маршрута доставки груза;
- ход процесса доставки груза.

Каждый из этих объектов-значений можно с легкостью заменить в рассматриваемом здесь корневом агрегате Cargo. Рассмотрим варианты сценариев и обоснования того, почему выбраны именно эти характеристики как объекты-значения, а не как сущности, поскольку это весьма важное решение при моделировании предметной области (домена):

- при новом заказе груза мы имеем новую спецификацию маршрута, пустой план маршрута доставки, а информация о ходе доставки отсутствует;
- при назначении плана маршрута доставки груза пустой объект-значение плана маршрута доставки заменяется сформированным объектом плана;

- при перемещении груза между несколькими портами в соответствии с планом маршрута доставки объект-значение хода процесса доставки груза обновляется и изменяется в корневом агрегате;
- наконец, если заказчик изменяет определение пункта назначения или предельный срок доставки груза, то меняется спецификация маршрута, назначается новый план маршрута и обновляется ход процесса доставки груза.

В каждом из описанных выше вариантов сценариев становится вполне понятно, что рассматриваемые объекты должны изменяться в корневом агрегате, поэтому их необходимо моделировать как объекты-значения.

Реализация объектов-значений подразумевает следующие аспекты:

- реализацию класса объекта-значения;
- определение отношений объект-значение–агрегат;
- создание объекта-значения;
- персистентность объекта-значения.

Реализация класса объекта-значения

Объекты-значения реализуются как встраиваемые объекты JPA с использованием стандартной аннотации `@Embeddable`, предоставляемой JPA.

> **ПРИМЕЧАНИЕ** Поскольку объекты-значения не обладают собственной идентичностью, они не имеют какого-либо первичного идентификатора.

В листинге 3.11 показана реализация объектов-значений `RouteSpecification`, `Itinerary` и `Delivery`, реализованных как встраиваемые объекты JPA.

Листинг 3.11 ❖ Объекты-значения, связанные с доставкой груза

```
@Embeddable
public class RouteSpecification implements Serializable {
  @ManyToOne
  @JoinColumn( name = "spec_origin_id", updatable = false )
  private Location origin;
  @ManyToOne
  @JoinColumn( name = "spec_destination_id" )
  private Location destination;
  @Temporal( TemporalType.DATE )
  @Column( name = "spec_arrival_deadline" )
  @NotNull
  private LocalDate arrivalDeadline;
}

@Embeddable
public class Delivery implements Serializable {
  public static final LocalDate ETA_UNKOWN = null;
  public static final HandlingActivity NO_ACTIVITY = new HandlingActivity();
  @Enumerated( EnumType.STRING )
  @Column( name = "transport_status" )
  @NotNull
  private TransportStatus transportStatus;
  @ManyToOne
  @JoinColumn( name = "last_known_location_id" )
  private Location lastKnownLocation;
```

```
  @ManyToOne
  @JoinColumn( name = "current_voyage_id" )
  private Voyage currentVoyage;
  @NotNull
  private boolean misdirected;
  private LocalDate eta;
  @Embedded
  private HandlingActivity nextExpectedActivity;
  @Column( name = "unloaded_at_dest" )
  @NotNull
  private boolean isUnloadedAtDestination;
  @Enumerated( EnumType.STRING )
  @Column( name = "routing_status" )
  @NotNull
  private RoutingStatus routingStatus;
  @Column( name = "calculated_at" )
  @NotNull
  private LocalDateTime calculatedAt;
  @ManyToOne
  @JoinColumn( name = "last_event_id" )
  private HandlingEvent lastEvent;
}
```

```
@Embeddable
public class Itinerary implements Serializable {
  public static final Itinerary EMPTY_ITINERARY = new Itinerary();
  @OneToMany( cascade = CascadeType.ALL, orphanRemoval = true )
  @JoinColumn( name = "booking_id" )
  @OrderBy( "loadTime" )
  @Size( min = 1 )
  private List<Leg> legs = Collections.emptyList();
}
```

Отношения между объектом-значением и агрегатом

Объекты-значения не могут существовать без корневого агрегата, но, поскольку они не имеют идентификатора, их можно с легкостью изменять в любом экземпляре агрегата.

Связи (отношения) между объектами-значениями и агрегатами реализуются с использованием аннотации `@Embedded`, предоставляемой JPA.

В листинге 3.12 показаны объекты-значения `RouteSpecification`, `Itinerary` и `Delivery`, связанные с корневым агрегатом Cargo как встроенные объекты.

Листинг 3.12 ❖ Объекты-значения корневого агрегата Cargo

```
@Embedded
private RouteSpecification routeSpecification;
@Embedded
private Itinerary itinerary;
@Embedded
private Delivery delivery;
```

Создание и персистентность объекта-значения

Формирование и обеспечение персистентности объектов-значений всегда осуществляется только совместно с соответствующим корневым агрегатом, когда эти операции выполняются в корневом агрегате.

В тот момент времени, когда происходит новый заказ груза, соответствующий агрегат не содержит назначенного ему плана доставки, и если мы попытаемся получить ход процесса доставки, то он окажется пустым, так как пока еще не был определен маршрут. Обратите особое внимание на то, как отображаются эти бизнес-концепции в корневом агрегате в листинге 3.13. Агрегату Cargo изначально присваивается пустой план маршрута доставки груза и мгновенный снимок состояния доставки на основе пустой хронологии транспортной обработки груза.

В листинге 3.13 показано, как создаются объекты-значения `Itinerary` и `Delivery` при создании корневого агрегата Cargo.

Листинг 3.13 ❖ Создание состояния объектов-значений

```
public Cargo( BookingId bookingId, RouteSpecification routeSpecification ) {
  this.bookingId = bookingId;
  this.origin = routeSpecification.getOrigin();
  this.routeSpecification = routeSpecification;
  this.itinerary = Itinerary.EMPTY_ITINERARY; // Пустой объект Itinerary, так как маршрут
                                              // доставки груза пока еще не назначен.
  this.delivery = new Delivery( this.routeSpecification, this.itinerary,
                                HandlingHistory.EMPTY );
            // Мгновенный снимок хода процесса доставки получен на основе пустой хронологии
            // транспортной обработки груза, так как это новый заказ груза.
}
```

Обобщенная схема рассматриваемой здесь реализации объекта-значения с использованием платформы Jakarta EE показана на рис. 3.12.

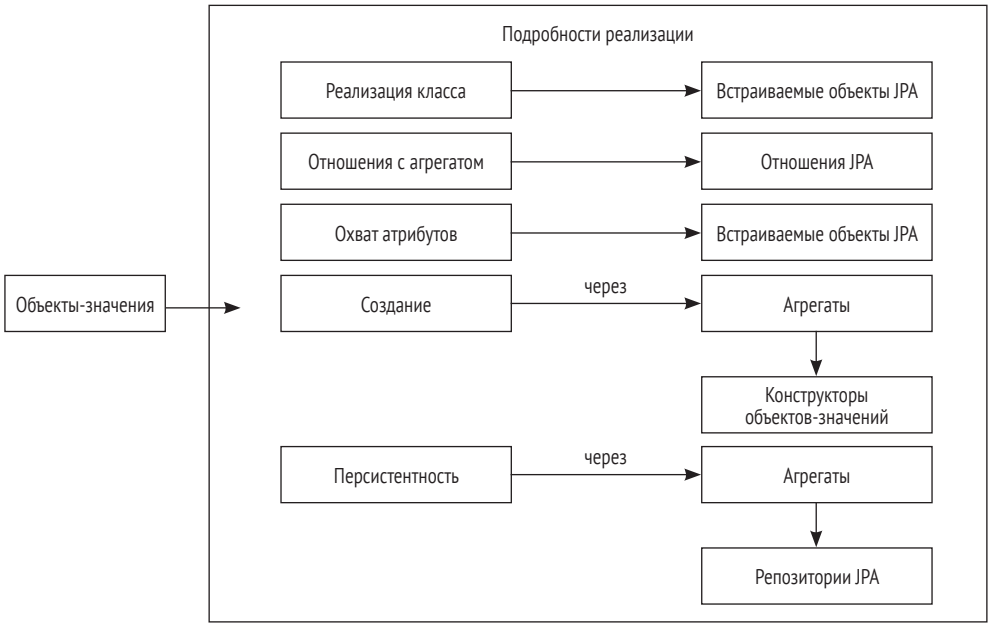

Рис. 3.12 ❖ Общая схема реализации объекта-значения

Правила предметной области (домена)

Правила предметной области (домена) помогают агрегату выполнять любой тип бизнес-логики в области видимости ограниченного контекста. Обычно эти правила дополняют информацию о состоянии агрегата, но сами по себе не определяют изменения в его состоянии. Такие правила представляют новые изменения состояния для сервисов приложения, которые отслеживают изменения состояния, чтобы выполнить соответствующие действия. Как уже было отмечено в предыдущей главе, правила предметной области (домена) могут существовать как в модели предметной области (домена), так и вне ее (на уровне сервисов).

Бизнес-правило (в модели предметной области (домена)) в обобщенном случае находит свое место в реализации объекта-значения как внутренняя закрытая (private) подпрограмма (метод). Продемонстрируем это на примере.

Корневой агрегат Cargo всегда связан с объектом-значением `Delivery`. При любом из возможных трех изменений, а именно когда определяется новая подпрограмма для груза, назначается маршрут для груза, выполняется транспортная обработка груза, необходимо непременно пересчитать ход процесса доставки. Рассмотрим реализацию конструктора объекта-значения `Delivery`, приведенную в листинге 3.14.

Листинг 3.14 ❖ Объект-значение Delivery

```
public Delivery( HandlingEvent lastEvent, Itinerary itinerary,
                 RouteSpecification routeSpecification ) {
  this.calculatedAt = new Date();
  this.lastEvent = lastEvent;
  this.misdirected = calculateMisdirectionStatus( itinerary );
  this.routingStatus = calculateRoutingStatus( itinerary, routeSpecification );
  this.transportStatus = calculateTransportStatus();
  this.lastKnownLocation = calculateLastKnownLocation();
  this.currentVoyage = calculateCurrentVoyage();
  this.eta = calculateEta( itinerary );
  this.nextExpectedActivity = calculateNextExpectedActivity( routeSpecification, itinerary );
  this.isUnloadedAtDestination = calculateUnloadedAtDestination( route Specification );
}
```

Эти вычисления представляют собой правила предметной области (домена), которые исследуют текущее состояние соответствующего агрегата для определения следующего состояния этого агрегата.

Как уже было отмечено ранее, правило предметной области (домена) для своего выполнения полагается исключительно на существующее состояние агрегата. В том случае, когда такое правило требует дополнительных данных помимо состояния агрегата, правило предметной области (домена) должно быть перемещено на уровень сервисов.

Обобщенная схема рассматриваемой здесь реализации правила предметной области (домена) показана на рис. 3.13.

Рис. 3.13 ❖ Общая схема реализации правила предметной области (домена)

Команды

Команда (command) в ограниченном контексте – это любая операция, которая изменяет состояние агрегата. Платформа Java EE не предлагает какие-либо специализированные средства для обозначения команд (операций), поэтому в рассматриваемой здесь реализации команды распределяются между сервисами приложения и моделью предметной области (домена). Модель предметной области (домена) частично изменяет состояние агрегата, а сервисы приложения постоянно вносят изменения.

Команда Change destination (Изменение пункта назначения) предполагает присваивание корневому агрегату Cargo новой спецификации маршрута и нового состояния процесса доставки.

В листинге 3.15 показан небольшой фрагмент реализации одной операции в корневом агрегате Cargo.

Листинг 3.15 ❖ Пример команды в корневом агрегате Cargo

```
public void specifyNewRoute( RouteSpecification routeSpecification ) {
  Validate.notNull(routeSpecification, "Route specification is required");
  this.routeSpecification = routeSpecification;
  this.delivery = delivery.updateOnRouting( this.routeSpecification,
                                            this.itinerary);
}
```

В листинге 3.16 показана реализация одной из команд в сервисах приложения ограниченного контекста Booking (заказ груза).

Листинг 3.16 ❖ Пример команды в корневом агрегате Cargo

```
public void changeDestination( BookingId bookingId, UnLocode unLocode ) {
  Cargo cargo = cargoRepository.find( bookingId );
  Location newDestination = locationRepository.find( unLocode );
  RouteSpecification routeSpecification = new RouteSpecification( cargo.getOrigin(),
                      newDestination, cargo.getRouteSpecification().getArrivalDeadline() );
  cargo.specifyNewRoute(routeSpecification);     // Обращение к модели домена.
  cargoRepository.store(cargo);                  // Сохранение состояния.
}
```

Обобщенная схема реализации команды показана на рис. 3.14.

Рис. 3.14 ❖ Общая схема реализации команды

Запросы

Запрос (Query) в ограниченном контексте – это любая операция, которая возвращает состояние агрегата. JPA предоставляет именованные запросы (Named Queries), которые можно применять в JPA-сущности для запроса состояния соответствующего агрегата. Репозитории JPA могут использовать именованные запросы для получения состояния агрегата.

В листинге 3.17 показано использование именованных запросов в корневом агрегате Cargo. Именованные запросы применяются для поиска всех грузов, для поиска груза по заданному идентификатору заказа (booking ID), а также для получения всех идентификаторов заказа.

Листинг 3.17 ❖ Именованные запросы в корневом агрегате Cargo

```
@Entity
@NamedQueries( {
  @NamedQuery( name = "Cargo.findAll", query = "Select c from Cargo c"),
  @NamedQuery( name = "Cargo.findByBookingId",
               query = "Select c from Cargo c where c.bookingId = :bookingId" ),
  @NamedQuery( name = "Cargo.getAllBookingIds",
               query = "Select c.bookingId from Cargo c" ) } )
public class Cargo implements Serializable {}
```

В листинге 3.18 показано использование именованного запроса `findBookingId` в репозитории JPA Cargo.

Листинг 3.18 ❖ Использование именованного запроса в корневом агрегате Cargo

```
@Override
public Cargo find( BookingId bookingId ) {
  Cargo cargo;
  try {
    cargo = entityManager.createNamedQuery( "Cargo.findByBookingId", Cargo.class )
            .setParameter( "bookingId", bookingId )
            .getSingleResult();
  } catch( NoResultException e ) {
      logger.log( Level.FINE, "Find called on non-existant Booking ID.", e );
      cargo = null;
  }
  return cargo;
}
```

Обобщенная схема реализации запроса показана на рис. 3.15.

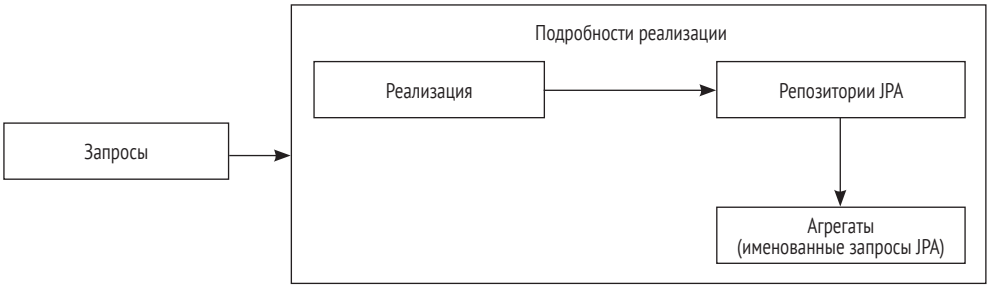

Рис. 3.15 ❖ Общая схема реализации запроса

Реализация сервисов предметной области с использованием Jakarta EE

Как уже было отмечено ранее, существуют три типа поддерживающих сервисов, которые требуются для основной модели предметной области (домена).

Входящие сервисы

Входящие сервисы (Inbound Services; или входящие адаптеры – Inbound Adaptors – в соответствии с обозначением в гексагональном архитектурном шаблоне) действуют как наиболее удаленный от ядра шлюз, обслуживающий основную модель предметной области (домена).

В приложении Cargo Tracker реализовано два типа входящих сервисов на основе типа пользователей модели предметной области (домена):

- HTTP API, реализованные с использованием RESTful веб-сервисов;
- собственные веб-API, реализованные с использованием JSF (JavaServer Faces) Managed Beans.

Можно было бы реализовать дополнительные входящие сервисы/адаптеры, классифицированные по типу поддерживаемого протокола, например входящий сервис на основе WebSocket для обновлений в реальном времени или входящий сервис на основе файлов для комплексных пакетных выгрузок. Все эти протоколы должны быть смоделированы как часть входящих сервисов.

Рассмотрим примеры каждого типа входящих сервисов в приложении Cargo Tracker и подробно изучим их реализацию с использованием платформы Jakarta EE.

RESTful API

Платформа Jakarta EE предоставляет функциональные возможности для реализации прикладных программных интерфейсов RESTful API с использованием спецификации JAX-RS. Эта спецификация является одной из наиболее широко применяемых на платформе Jakarta EE.

Пример использования RESTful API на основе спецификации JAX-RS в приложении Cargo Tracker показан в листинге 3.19.

Листинг 3.19 ❖ Пример использования RESTful API

```
package com.practicalddd.cargotracker.handling.interfaces.rest;
@Path( "/handling" )
public class HandlingService {
  @POST
  @Path( "/reports" )
  @Consumes( "application/json" )
  public void submitReport( @NotNull @Valid HandlingReport handlingReport ) {
  }
}
```

Этот интерфейс RESTful API предъявлен как часть ограниченного контекста Handling (транспортная обработка груза) и отвечает за обработку груза в транзитном порту. Путь к этому интерфейсу обозначен как `/handling/reports`. API использует структуру JSON и запрос POST. Это обычные конструкции RESTful, поддерживаемые спецификацией JAX-RS.

Собственные веб-API

Второй тип входящих сервисов реализован с помощью собственных веб-API (Native Web API). Веб-интерфейс администратора (Cargo Admin Web Interface) – это тонкий интерфейс на основе браузера, реализованный с использованием JavaServer Faces (JSF). Фактически это стандарт для создания HTML-приложений на платформе Jakarta EE.

Интерфейс JSF основан на широко распространенном шаблоне проектирования MVC (Model View Controller – Модель Представление Контроллер) с моделью, реализованной с использованием JSF Managed Beans на основе CDI (Component Dependency Injection – механизм инъекции зависимостей компонента). Модель работает как уровень входящего сервиса для веб-интерфейсов.

Пример использования собственных веб-API (Native Web API) с применением JSF/CDI в приложении Cargo Tracker показан в листинге 3.20.

Листинг 3.20 ❖ Пример использования веб-API

```
package com.practicalddd.cargotracker.booking.interfaces.web;

@Named // Имя компонента bean.
@RequestScoped // Область видимости компонента bean.
public class CargoAdmin {
  public String bookCargo() {
         // Вызов модели домена для заказа нового груза.
  }
}
```

Класс `CargoAdmin` реализован как веб-API с использованием JSF и CDI Beans. В этот класс включен набор операций (например, `bookCargo`) для веб-интерфейса администратора Cargo Admin Web Interface, с которым работает сотрудник, выполняющий разнообразные операции (например, заказ груза). Веб-интерфейс

администратора Cargo может вызывать эти операции в используемых здесь CDI Beans.

Обобщенная схема реализации входящих сервисов показана на рис. 3.16.

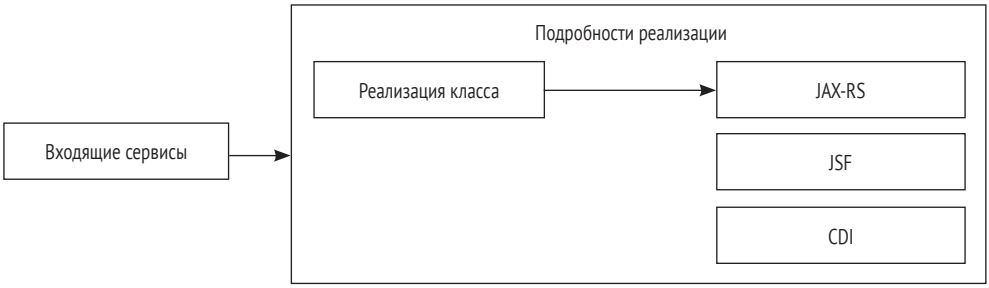

Рис. 3.16 ❖ Общая схема реализации входящих сервисов

Сервисы приложения

Сервисы приложения создаются с использованием компонентов CDI, доступных на платформе Jakarta EE. Выше мы уже обращались к теме использования CDI, но не углублялись в подробности. Теперь мы более подробно рассмотрим этот аспект.

Реализация концепции CDI впервые была представлена в версии Java EE 6.0, и компоненты CDI эффективно заменили компоненты EJB в качестве действительного инструментального средства на платформе Jakarta EE для создания бизнес-компонентов. CDI управляет жизненным циклом и взаимодействиями компонентов с поддержкой типобезопасной инъекции зависимостей. Кроме того, CDI предоставляет полноценную программную среду SPI (Service Provider Framework), позволяющую создавать переносимые расширения и интегрировать их в платформу Jakarta EE.

Создание сервисов приложения с использованием CDI подразумевает выполнение нескольких описанных ниже шагов. Рассмотрим сервисы приложения Cargo Booking на конкретном примере:

- создание обычного интерфейса Java с операциями, которые предоставляют сервисы приложения. Это показано в листинге 3.21:

  Листинг 3.21 ❖ Интерфейс сервисов приложения Booking

```
package com.practicalddd.cargotracker.booking.application;

public interface BookingService {
  BookingId bookNewCargo( UnLocode origin, UnLocode destination,
                          LocalDate arrivalDeadline);
  List<Itinerary> requestPossibleRoutesForCargo( BookindId bookingId );
  void assignCargoToRoute( Itinerary itinerary, BookingId bookingId );
  void changeDestination( BookingId bookingId, UnLocode unLocode );
}
```

- реализация интерфейса, размеченного с помощью специализированных аннотаций CDI. Аннотации CDI обычно используются для определения

области видимости компонента вместе с его именем (в том случае, если существует несколько реализаций конкретного интерфейса).
Здесь представлена реализация интерфейса сервиса Booking (заказ груза), которому назначается область видимости приложения, т. е. отдельный экземпляр для всего приложения в целом. Это показано в листинге 3.22:

Листинг 3.22 ❖ Реализация сервисов приложения Booking

```
package com.practicalddd.cargotracker.booking.application.internal;

@ApplicationScoped // Аннотация CDI для определения области видимости.
public class  DefaultBookingService implements BookingService {
  BookingId bookNewCargo( UnLocode origin, UnLocode destination,
                          LocalDate arrivalDeadline ) {
            // Здесь приводится реализация.
  }
  List<Itinerary> requestPossibleRoutesForCargo( BookindId bookingId ) {
            // Здесь приводится реализация.
  }
  void assignCargoToRoute( Itinerary itinerary, BookingId bookingId ) {
            // Здесь приводится реализация.
  }
  void changeDestination( BookingId bookingId, UnLocode unLocode ) {
            // Здесь приводится реализация.
  }
}
```

- в сервисах приложения будет содержаться набор зависимостей; например, может потребоваться доступ к репозиторию инфраструктуры классов для извлечения подробной информации об агрегате как часть конкретной операции. В листинге 3.6 это было показано схематично, где в сервисах приложения использовался класс `CargoRepository` для загрузки агрегата Cargo. В классе сервисов приложения представлена зависимость от класса `CargoRepository` с помощью аннотации CDI «Inject». Это показано в листинге 3.23:

Листинг 3.23 ❖ Зависимости сервисов приложения Booking

```
package com.practicalddd.cargotracker.booking.application.internal;

@Inject          // Инъекция зависимости от инфраструктуры репозитория Cargo.
private CargoRepository cargoRepository;

@ApplicationScoped // Аннотация CDI для определения области видимости.
public class DefaultBookingService implements BookingService {
  @Override
  public List<Itinerary> requestPossibleRoutesForCargo( BookingId bookingId ) {
    Cargo cargo = cargoRepository.find( bookingId ); // Использование репозитория
                                                     // Cargo.
            // Далее здесь приводится реализация.
  }
}
```

Обобщенная схема реализации инъекции зависимостей показана на рис. 3.17.

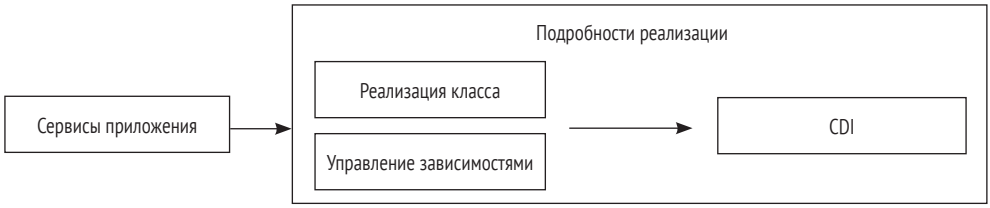

Рис. 3.17 ❖ Общая схема реализации инъекции зависимостей

Сервисы приложения: события

Как уже отмечалось ранее, события предметной области (домена) в истинном духе предметно-ориентированного проектирования должны генерироваться агрегатом. Поскольку платформа Jakarta EE не предоставляет механизм генерации событий предметной области (домена) агрегатом, необходимо переместить эту функцию на уровень сервисов приложения.

Инфраструктура события основана на механизме CDI Events. Впервые представленный в версии CDI 2.0, этот механизм предлагает весьма аккуратно выполненную реализацию модели оповещения и отслеживания событий. Это позволяет обеспечить низкий уровень связности между различными ограниченными контекстами, что помогает получить требуемое проектное решение действительно модульного монолитного приложения.

Модель обработки событий CDI показана на рис. 3.18.

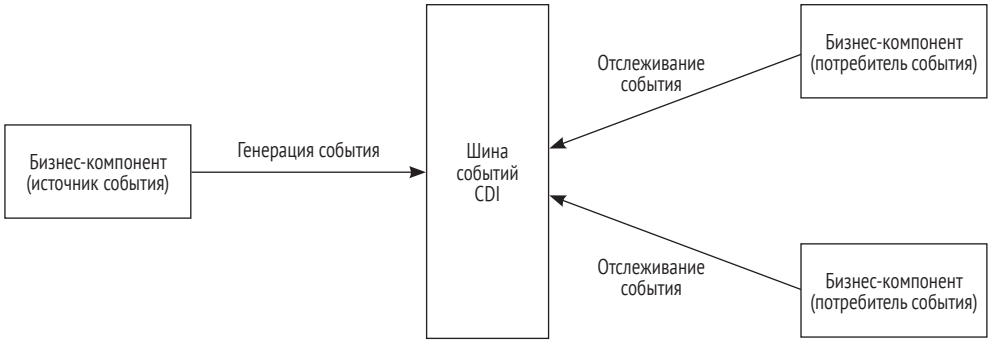

Рис. 3.18 ❖ Модель обработки событий CDI

Шина событий CDI не является специализированной шиной обработки событий, это внутренняя реализация шаблона проектирования Observer (наблюдателя), выполненная в контейнере с расширенной поддержкой, включая поддержку наблюдателей транзакций (Transactional Observers), наблюдателей условий (Conditional Observers), упорядочения (Ordering), а также синхронных и асинхронных событий.

Рассмотрим подробнее реализацию этого механизма в приложении Cargo Tracker. Ограниченный контекст Handling (транспортная обработка груза) генерирует событие «Груз проинспектирован» (Cargo Inspected) каждый раз, когда выполняется проверка груза как одна из операций транспортной обработки. Ограниченный контекст Tracking (отслеживание груза) отслеживает это событие и соответствующим образом обновляет данные об отслеживании процесса доставки груза.

Общая схема потока событий показана на рис. 3.19.

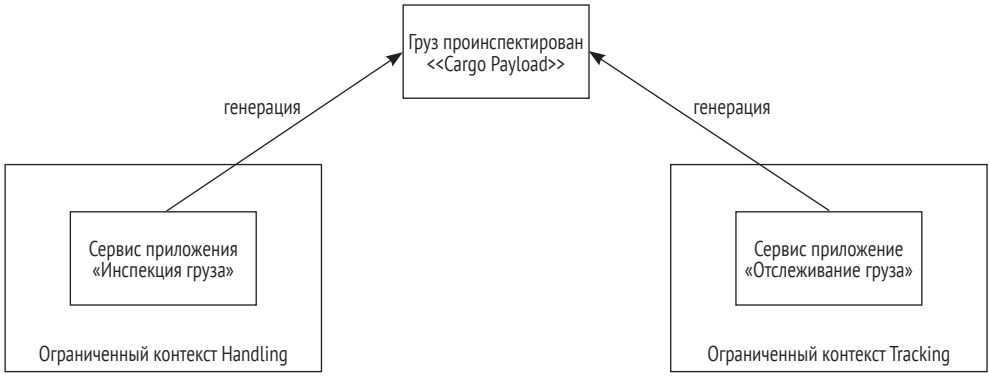

Рис. 3.19 ❖ Потоки событий
между ограниченными контекстами Handling и Tracking

С точки зрения реальной реализации выполняются следующие шаги.

- Создание класса события (с использованием механизма стереотипов). В листинге 3.24 показано создание класса события `CargoInspected`.

Листинг 3.24 ❖ Класс события CargoInspected с использованием механизма стереотипов

```
package com.practicalddd.cargotracker.handling.infrastructure.events.cdi;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.inject.Qualifier;

@Qualifier // Аннотация стереотипа.
@Retention( RUNTIME ) // Аннотация стереотипа.
@Target( {FIELD, PARAMETER} ) // Аннотация стереотипа.
public @interface CargoInspected { // Событие.
}
```

- Генерация события.
  В листинге 3.25 показана генерация события «Груз проинспектирован» из сервисов приложения, в данном случае это сервисы приложения Cargo Inspected (Груз проинспектирован).

Листинг 3.25 ❖ Генерация события «Груз проинспектирован»

```
package com.practicalddd.cargotracker.handling.application.internal;

import javax.enterprise.event.Event;
import javax.inject.Inject;
// Импортирование стереотипного класса события.
import com.practicalddd.cargotracker.infrastructure.events.cdi.CargoInspected;

public class DefaultCargoInspectionService implements CargoInspectionService {
@Inject
@CargoInspected
private Event<Cargo> cargoInspected; // Событие, которое будет генерироваться.
                                     // Полезная нагрузка - агрегат Cargo.
  /**
   * Метод, который будет выполнять инспекцию груза и генерировать
   * соответствующее событие.
   */
  public void inspectCargo( BookingId bookingId ) {
    // Загрузка объекта Cargo (груз).
    Cargo cargo = cargoRepository.find( bookingId );
    // Выполнение процедуры инспекции.
    // Генерация события после инспекции.
    cargoInspected.fire( cargo );
  }
}
```

- Отслеживание события.
  Сервис отслеживания груза (Cargo Tracking) в ограниченном контексте Tracking отслеживает сгенерированное событие и соответствующим образом обновляет данные об отслеживании процесса доставки груза. Это показано в листинге 3.26.

Листинг 3.26 ❖ Отслеживание события «Груз проинспектирован»

```
package com.practicalddd.cargotracker.tracking.application.internal;

import javax.enterprise.event.Event;
import javax.inject.Inject;
import com.practicalddd.cargotracker.infrastructure.events.cdi.CargoInspected;

public class DefaultTrackingService implements TrackingService {
@Inject
@CargoInspected
private Event<Cargo> cargoInspected; // Подписка на конкретное событие.
  /**
   * Метод, который отслеживает событие CDI и обрабатывает полезную нагрузку.
   */
  public void onCargoInspected( @Observes @CargoInspected Cargo cargo ) {
    // Обработка события.
  }
}
```

Обобщенная схема реализации обработки событий показана на рис. 3.20.

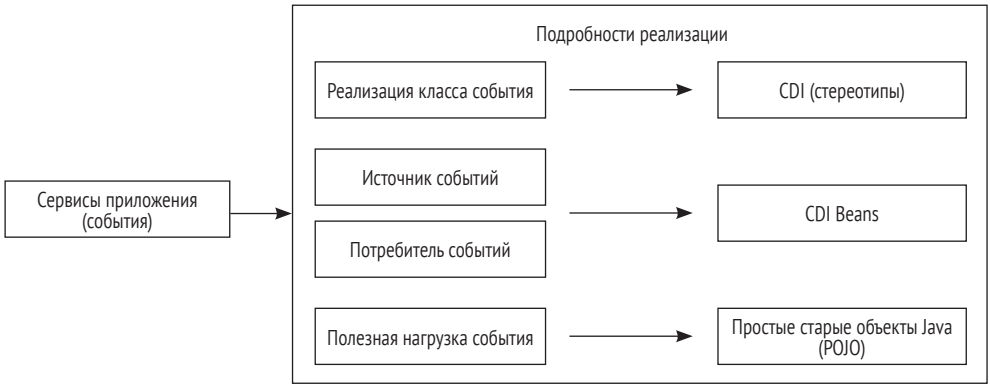

Рис. 3.20 ❖ Общая схема реализации механизма обработки событий на уровне сервисов приложения

Исходящие сервисы

В монолитном приложении Cargo Tracker исходящие сервисы используются главным образом для обмена данными с репозиторием базы данных более низкого уровня. Исходящие сервисы реализуются как классы Repository и являются частью уровня инфраструктуры.

Классы Repository создаются с использованием JPA и применяют механизм CDI для управления своим жизненным циклом. JPA предоставляет управляемый ресурс `EntityManager`, который позволяет абстрагироваться от подробностей конфигурации конкретной базы данных (например, Datasource).

В общем случае класс Repository является в определенном смысле оберткой для конкретного агрегата и имеет дело со всеми операциями базы данных для этого агрегата, включая следующие:

- обеспечение персистентности нового агрегата и его отношений;
- обновление агрегата и его отношений;
- выполнение запросов агрегата и его отношений.

В листинге 3.27 показан пример класса репозитория `JPACargoRepository`.

Листинг 3.27 ❖ Реализация класса репозитория для объекта Cargo

```
package com.practicalddd.cargotracker.booking.infrastructure.persistence.jpa;

// Аннотации JPA.
import javax.enterprise.context.ApplicationScoped;
import javax.persistence.EntityManager;
import javax.persistence.NoResultException;
import javax.persistence.PersistenceContext;

@ApplicationScoped
public class  JpaCargoRepository implements CargoRepository, Serializable {
@PersistenceContext
  private EntityManager entityManager; // Управляемый ресурс, используемый классом
                                       // репозитория для взаимодействия с базой данных.
// Сохранение объекта Cargo.
```

```
@Override
  public void store( Cargo cargo ) {
    entityManager.persist( cargo );
  }
// Поиск всех объектов Cargo. Используется именованный запрос, определенный в корневом
// агрегате Cargo. См. также Queries (запросы).
@Override
  public List<Cargo> findAll() {
    return entityManager.createNamedQuery( "Cargo.findAll", Cargo.class ).getResultList();
  }
// Поиск конкретного объекта Cargo. Используется именованный запрос, определенный
// в корневом агрегате.
@Override
  public Cargo find( BookingId bookingId ) {
    Cargo cargo;
    try {
      cargo = entityManager.createNamedQuery( "Cargo.findByBookingId", Cargo.class)
                           .setParameter( "bookingId", bookingId )
                           .getSingleResult();
    } catch( NoResultException e ) {
        logger.log( Level.FINE, "Find called on non-existant Booking ID.", e );
        cargo = null;
    }
    return cargo;
  }
}
```

Обобщенная схема реализации исходящего сервиса показана на рис. 3.21.

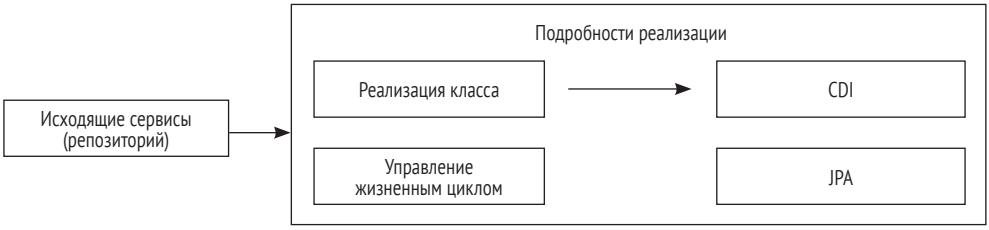

Рис. 3.21 ❖ Общая схема реализации исходящего сервиса

На этом завершается обзор реализации Cargo Tracker как модульного монолитного приложения с использованием платформы Jakarta EE 8.

Общая схема реализации

Теперь у нас есть полная реализация предметно-ориентированного проектирования для монолитного приложения Cargo Tracker с различными артефактами предметно-ориентированного проектирования, реализованными с использованием соответствующих спецификаций, предоставленных платформой Java EE.

Общая схема реализации показана на рис. 3.22.

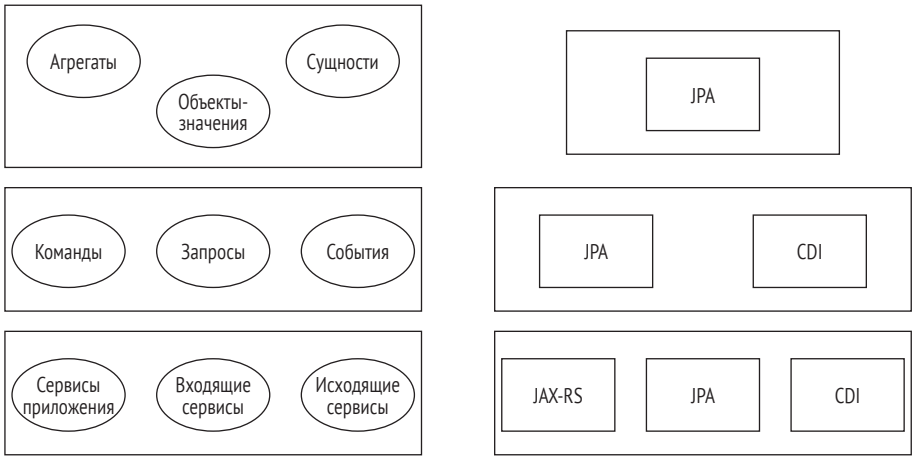

Рис. 3.22 ❖ Общая схема реализации артефактов предметно-ориентированного проектирования с использованием платформы Java EE

Резюме

Краткое обобщение содержимого этой главы:

- в начале главы была подробно описана платформа Jakarta EE и предоставляемые ею разнообразные функциональные возможности;
- затем было обосновано решение о реализации Cargo Tracker как модульного монолитного приложения с использованием предметно-ориентированного проектирования;
- в остальной части главы подробно рассматривалась разработка различных артефактов предметно-ориентированного проектирования – сначала модель предметной области (домена), затем поддерживающие сервисы модели предметной области с использованием технологий и методов, предлагаемых платформой Jakarta EE.

Глава 4

Проект Cargo Tracker: Eclipse MicroProfile

Краткий обзор предыдущих глав:

- отслеживание доставки груза определено как основная область задачи, т. е. основная предметная область (домен), а приложение Cargo Tracker – как решение для этой области задачи;
- определены различные поддомены/ограниченные контексты для приложения Cargo Tracker;
- подробно описана модель предметной области (домена) для каждого определенного ограниченного контекста, в том числе были определены агрегаты, сущности, объекты-значения и правила предметной области (домена);
- определены поддерживающие сервисы предметной области (домена), необходимые для ограниченных контекстов;
- определены различные операции в ограниченных контекстах (команды, запросы, события и саги);
- описана реализация монолитной версии приложения Cargo Tracker с использованием платформы Jakarta EE.

В этой главе подробно рассматривается второй вариант реализации предметно-ориентированного проектирования для приложения Cargo Tracker с использованием новой платформы Eclipse MicroProfile. Приложение Cargo Tracker будет проектироваться с применением архитектуры, основанной на микросервисах. Как и ранее, артефакты предметно-ориентированного проектирования будут отображаться в соответствующие объекты реализации, доступные на платформе Eclipse MicroProfile.

В первую очередь необходимо сделать обзор платформы Eclipse MicroProfile.

Платформа Eclipse MicroProfile

После того как архитектура микросервисов стала быстро внедряться на промышленных предприятиях, сразу же возникла необходимость в удовлетворении соответствующих требований на платформе Java EE. Положение осложнялось слишком длительным интервалом времени между выпусками версий, а также тем, что платформа Java EE была в большей степени сосредоточена на

привычных монолитных приложениях, поэтому группа компаний, поддерживающих в то время платформу Java EE, решила сформировать более оптимизированную платформу, подходящую для микросервисов, с ускоренным циклом выпуска версий.

Эта новая платформа, названная MicroProfile, впервые появилась в 2016 году, а после объединения с организацией Eclipse Foundation получила окончательное имя Eclipse MicroProfile. Целью платформы MicroProfile было использование строгих основополагающих спецификаций платформы Java EE и расширение их с помощью набора спецификаций, специализированных для микросервисов и облачных сред. Таким образом, эта платформа позиционировалась как полноценная платформа поддержки микросервисов на мощной основе платформы Jakarta EE.

За весьма короткий промежуток времени платформа Eclipse MicroProfile получила широкое распространение и поддержку сообщества. Кроме того, участниками проекта было создано около девяти реализаций различных спецификаций. Эти спецификации были тщательно продуманы с учетом ныне существующих и будущих проблем и особенностей, присущих технологии микросервисов.

Стиль архитектуры микросервисов

Стиль архитектуры микросервисов весьма быстро стал основой для создания корпоративных (промышленных) приложений следующего поколения. Он обеспечивает независимость в течение полного процесса разработки программного обеспечения, а жизненный цикл развертывания и сдачи в эксплуатацию корпоративных приложений существенно ускоряется за счет увеличения скорости доставки программного обеспечения. Краткий обзор преимуществ микросервисов показан на рис. 4.1.

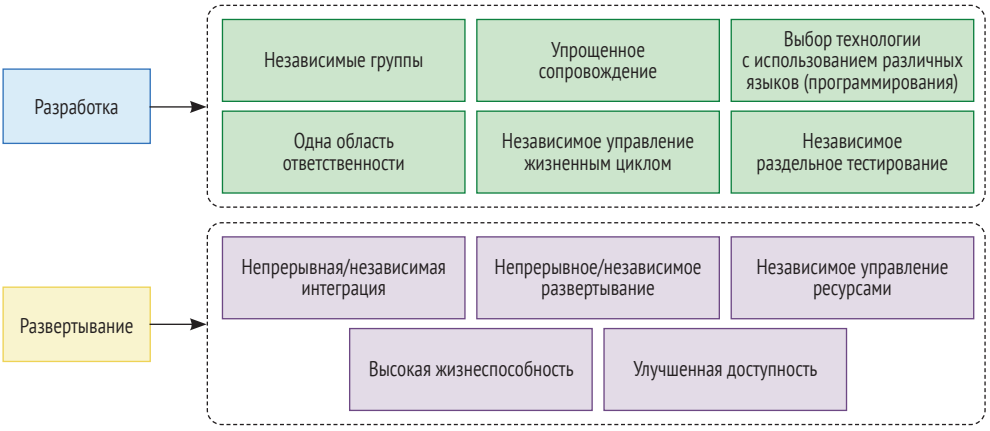

Рис. 4.1 ❖ Преимущества архитектуры на основе микросервисов

Но в архитектуре микросервисов существует и определенная группа сложностей. По своей сущности архитектура микросервисов является распределенной, поэтому сложности реализации возникают в следующих областях:

- управлении транзакциями;
- управлении данными;
- шаблонах уровня приложения (например, создании отчетов, службе аудита);
- архитектуре развертывания;
- распределенном тестировании.

Поскольку стиль архитектуры микросервисов становится все более широко распространенным, проблемы в перечисленных выше областях решаются различными способами с использованием программных рабочих сред с открытым исходным кодом, а также с использованием проприетарного программного обеспечения от компаний-участников группы поддержки платформы.

Платформа Eclipse MicroProfile: функциональные возможности

Платформа Eclipse MicroProfile предоставляет весьма прочную основу для проектирования и планирования перехода приложений на архитектуру микросервисов. В совокупности с предметно-ориентированным проектированием, предоставляющим четко определенные процессы и шаблоны для проектирования и разработки микросервисов, формируется мощная платформа для создания приложений на основе микросервисов.

Прежде чем перейти к подробному рассмотрению технических компонентов платформы MicroProfile, необходимо сформулировать требования к платформе микросервисов, как показано на рис. 4.2.

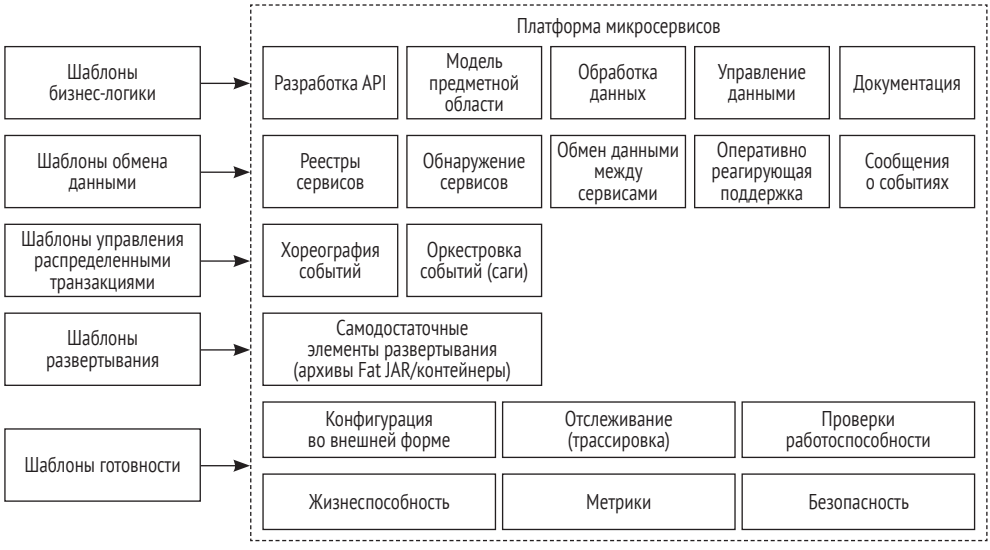

Рис. 4.2 ❖ Требования к полноценной платформе микросервисов

Эти требования разделены по нескольким областям, которые предъявляют собственные специфические требования к платформе микросервисов.

Набор предоставляемых платформой Eclipse MicroProfile спецификаций показан на рис. 4.3. Спецификации сгруппированы по двум категориям:

- основной набор, который помогает выполнить специализированные требования архитектуры микросервисов/облачной среды. Эти спецификации предоставляют решения в областях конфигурирования, проверок работоспособности, шаблонов обмена данными, мониторинга, безопасности и жизнеспособности;
- набор поддержки, который помогает выполнить обычные требования приложений вне зависимости от того, являются ли приложения монолитными или основаны на микросервисах. В эту категорию включены функциональные возможности для создания бизнес-логики, проектирования API, управления данными и обработки данных.

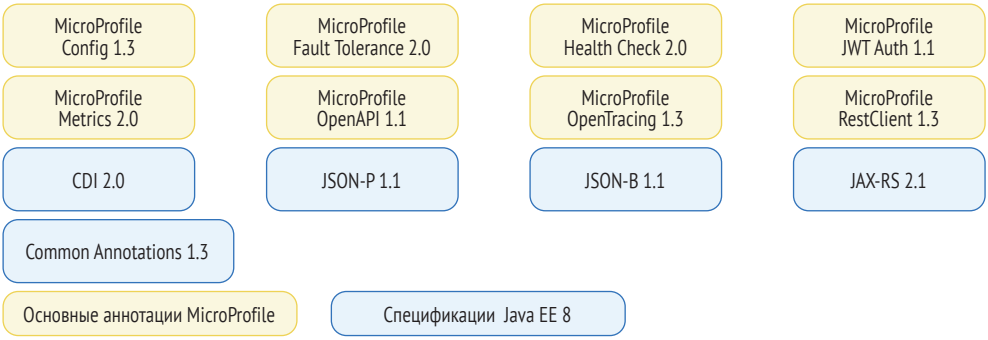

Рис. 4.3 ❖ Спецификации платформы Eclipse MicroProfile

В отличие от основной платформы Java EE/Jakarta EE в проекте MicroProfile нет разделения по профилям. Это просто единый набор спецификаций, который обязан реализовать любой участник группы поддержки платформы, чтобы обеспечить соответствие требованиям MicroProfile.

В табл. 4.1 приведен список участников проекта поддержки платформы, которые предлагают реализации спецификаций MicroProfile.

Таблица 4.1. Компании, реализующие спецификации Eclipse MicroProfile

Наименование реализации	Версия реализации
Helidon (Oracle)	MicroProfile 2.2
SmallRye (сообщество)	MicroProfile 2.2
Thorntail (Red Hat)	MicroProfile 3.0
Open Liberty (IBM)	MicroProfile 3.0
WebSphere Liberty (IBM)	MicroProfile 3.0
Payara Server (Payara Services)	MicroProfile 2.2
Payara Micro (Payara Services)	MicroProfile 2.2
TomEE (Apache)	MicroProfile 2.0
KumuluzEE (Kumuluz)	MicroProfile 2.2

Рассмотрим более подробно спецификации Eclipse MicroProfile. Сначала сделаем обзор основной группы спецификаций, затем перейдем к спецификациям группы поддержки.

Платформа MicroProfile: основные спецификации

Основные спецификации MicroProfile помогают реализовать набор технических требований, предъявляемых приложениями на основе микросервисов и предназначенных для работы в облачной среде. Этот набор спецификаций продуман и спроектирован так, чтобы позволить разработчикам без затруднений перейти к стилю микросервисов для реализации их функциональных возможностей.

С учетом требований технологии микросервисов отображение их реализаций в основных спецификациях показано прямоугольниками с заливкой на рис. 4.4.

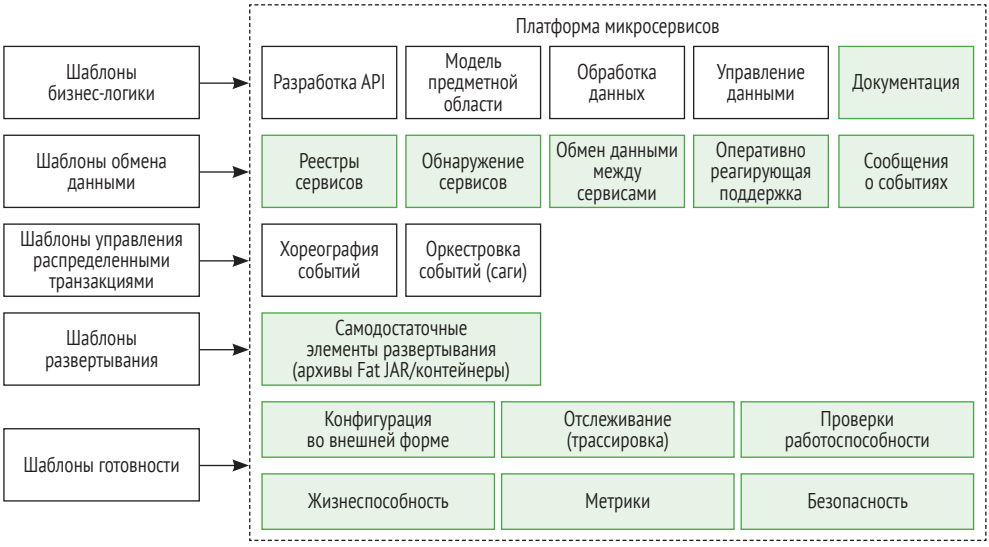

Рис. 4.4 ❖ Основные спецификации Eclipse MicroProfile и их связи с требованиями микросервисов

Теперь рассмотрим подробнее отдельные спецификации MicroProfile.

Конфигурация Eclipse MicroProfile

Спецификация конфигурации определяет простой в практическом применении механизм реализации конфигурации приложения, требуемый микросервисами. Для каждого микросервиса непременно потребуется некоторый тип конфигурации (например, места размещения ресурсов как URL других сервисов или связность баз данных, бизнес-конфигурации, флаги функциональных свойств). Конфигурационная информация также может быть различной в зависимости от среды развертывания микросервисов (например, среда разра-

ботки, среда тестирования, эксплуатационная среда). Артефакт микросервисов не должен изменяться для адаптации к различным шаблонам конфигурации.

Конфигурация MicroProfile определяет стандартный способ вовлечения и инъекции конфигурационной информации, требуемой микросервисами, без необходимости внесения изменений в пакет артефакта. Таким образом, предоставляется механизм инъекции конфигурации, заданной по умолчанию, в сочетании с механизмами замещения значений по умолчанию через внешние средства (переменные среды, переменные командной строки Java, переменные контейнера).

В дополнение к механизму инъекции конфигурационной информации спецификация MicroProfile Config также определяет стандартный способ реализации источников конфигурации (Configuration Sources), т. е. репозиторий, где хранится конфигурационная информация. Источниками конфигурации могут быть репозитории Git или базы данных.

Текущая версия этой спецификации – 1.3.

Проверка работоспособности Eclipse MicroProfile

Спецификация проверки работоспособности MicroProfile Health Check определяет стандартный механизм времени выполнения для определения состояния и видимости микросервисов. Этот механизм предназначен для использования в контейнерной среде через механизм межмашинного взаимодействия.

Текущая версия этой спецификации – 2.0.

Аутентификация Eclipse MicroProfile JWT Authentication

Спецификация Eclipse MicroProfile JSON Web Token (JWT) Authentication определяет стандартный механизм защиты для реализации процедур аутентификации и авторизации (RBAC – Role-Based Access Control) для конечных пунктов микросервисов с использованием механизма JSON Web Token (JWT).

Текущая версия этой спецификации – 1.1.

Метрики Eclipse MicroProfile

Спецификация MicroProfile Metrics определяет стандартный механизм микросервисов для генерации метрик, которые распознаются инструментальными средствами мониторинга.

Текущая версия этой спецификации – 2.0.

Eclipse MicroProfile OpenAPI

Спецификация MicroProfile OpenAPI определяет стандартный механизм генерации контрактов и документов, совместимых с OpenAPI, для микросервисов.

Текущая версия этой спецификации – 1.1.

Eclipse MicroProfile OpenTracing

Спецификация MicroProfile OpenTracing определяет стандартный механизм реализации распределенной трассировки в приложениях на основе микросервисов.

Текущая версия этой спецификации 1.3.

Eclipse MicroProfile Type Safe Rest Client

Спецификация MicroProfile OpenTracing определяет стандартный механизм реализации вызовов RESTful API между микросервисами.

Текущая версия этой спецификации – 1.3.

На этом завершается обзор набора спецификаций, предоставляемых Eclipse MicroProfile. Как можно видеть, эти спецификации весьма тщательно проработаны и предлагают исчерпывающий полноценный набор функциональных возможностей, позволяющий создавать приложения на основе микросервисов в полном соответствии со стандартами.

Eclipse MicroProfile: спецификации поддержки

Основной набор спецификаций позволяет реализовать главные функции микросервисов, в то время как спецификации поддержки помогают создавать функции бизнес-логики для микросервисов, т. е. модель предметной области (домена), интерфейсы API, механизмы обработки данных и управления данными.

С учетом требований технологии микросервисов отображение их реализаций в спецификациях поддержки показано прямоугольниками с заливкой желтым цветом на рис. 4.5.

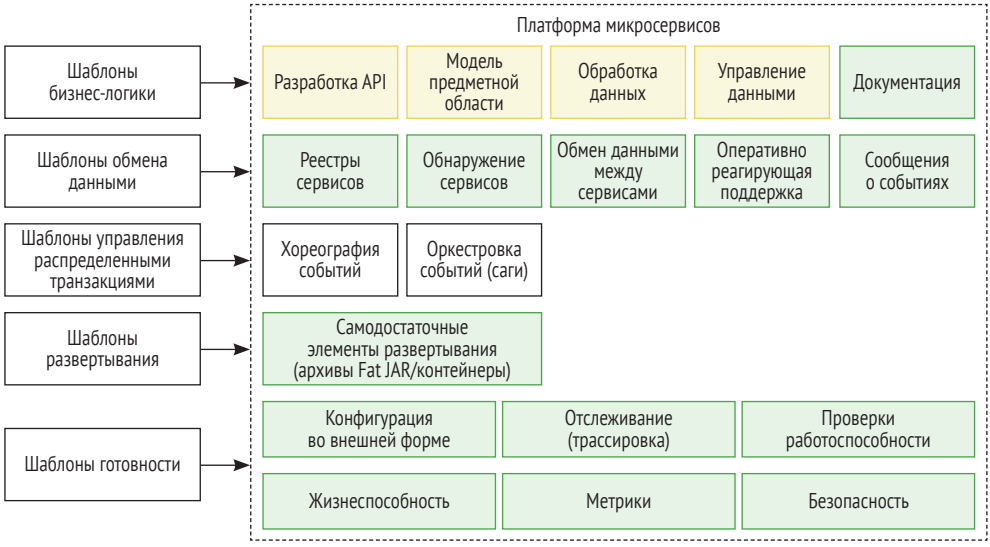

Рис. 4.5 ❖ Спецификации поддержки Eclipse MicroProfile

Теперь рассмотрим подробнее отдельные спецификации поддержки.

Context and Dependency Injection (CDI) for Java (2.0)

Как уже отмечалось в предыдущей главе, механизм контекстов и инъекции за-

висимостей (CDI) был представлен в спецификации Java EE для создания компонента и управления его зависимости через механизм инъекций. В настоящее время механизм CDI уже стал основополагающим элементом технологии почти для всех частей платформы, в то время как компоненты EJB постепенно выходят из употребления.

Самая последняя версия этой спецификации – 2.0.

Общие аннотации

Эта спецификация (Common Annotations) предоставляет набор аннотаций, иначе называемых маркерами (markers), которые помогают контейнеру времени выполнения реализовывать общие задачи (например, инъекции ресурсов, управление жизненным циклом).

Самая последняя версия этой спецификации – 1.3.

Java API for RESTful Web Services (JAX-RS)

Эта спецификация предоставляет разработчикам стандартный интерфейс Java API для реализации RESTful веб-сервисов. Еще одна широко известная спецификация, основной релиз самой последней версии которой предоставляет поддержку оперативно реагирующих клиентов (Reactive Clients) и событий на стороне сервера.

Самая последняя версия этой спецификации – 2.1.

Java API for JSON Binding

Это новая спецификация, представленная в версии Java EE 8 и подробно описывающая API, который предоставляет уровень связывания для преобразования объектов языка Java в сообщения JSON и наоборот.

Первая версия этой спецификации, разумеется, имеет номер 1.0.

Java API for JSON Processing

Эта спецификация предоставляет API, который можно использовать для доступа к объектам JSON и их обработки. Основной релиз самой последней версии этой спецификации на платформе Java EE 8 содержал разнообразные расширения и улучшения, как, например, JSON Pointer, JSON Patch, JSON Merge Patch и JSON Collectors.

Текущая версия этой спецификации – 1.1.

Как можно видеть, платформа не предоставляет прямую поддержку «из коробки» для управления распределенными транзакциями с использованием саг на основе оркестровки. Поэтому потребуется реализация распределенных транзакций с использованием хореографии событий (Event Choreography). Предполагается, что в будущих версиях платформы MicroProfile запланирована реализация шаблонов оркестровки саг как часть спецификации MicroProfile

LRA (Long Running Action – долговременная операция).

Итоговый обзор спецификаций Eclipse MicroProfile

На этом завершается общий обзор спецификаций платформы Eclipse MicroProfile. Это исчерпывающие спецификации, которые предоставляют почти все функциональные возможности и свойства, необходимые для создания промышленных (корпоративных) приложений, полностью соответствующих стилю архитектуры микросервисов. Кроме того, платформа предоставляет возможности расширения в тех случаях, когда существующих средств недостаточно для удовлетворения специфических потребностей промышленного предприятия.

Как и для платформы Jakarta EE, самым важным фактом является то, что это стандартные спецификации, подготовленные многочисленными участниками группы поддержки платформы. Это обеспечивает для промышленных предприятий большую вариативность при выборе конкретной платформы микросервисов.

Реализация Cargo Tracker: Eclipse MicroProfile

Следует напомнить, что целью этой главы является реализация Cargo Tracker как приложения на основе микросервисов с использованием предметно-ориентированного проектирования и платформы Eclipse MicroProfile.

Как часть процесса реализации мы будем использовать предметно-ориентированное проектирование в качестве основы при проектировании и разработке микросервисов. По ходу процесса реализации мы будем отображать и реализовывать артефакты предметно-ориентированного проектирования в соответствующие инструментальные средства, доступные на платформе Eclipse MicroProfile.

Двумя основными частями любой архитектуры на основе предметно-ориентированного проектирования являются модель предметной области (домена) и сервисы модели предметной области (домена). Это показано на рис. 4.6. В предыдущей главе было продемонстрировано применение предметно-ориентированного проектирования для реализации Cargo Tracker как монолитного приложения. В этой главе будет показано использование предметно-ориентированного проектирования для достижения назначенной здесь цели, т. е. реализации Cargo Tracker как приложения на основе микросервисов.

Здесь могут встречаться некоторые повторяющиеся фрагменты материала предыдущей главы из-за общности реализаций некоторых из артефактов предметно-ориентированного проектирования. Будем считать, что вы читаете эту главу, как если бы намеревались реализовать проект создания микросервисов с нуля с использованием платформы Eclipse MicroProfile.

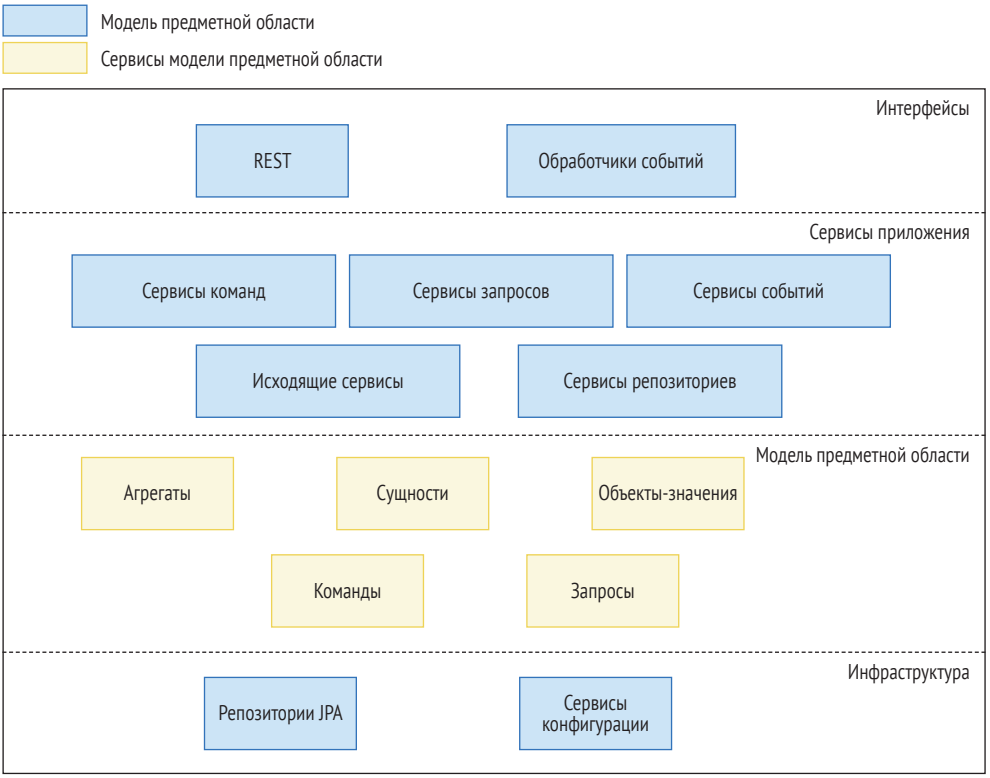

Рис. 4.6 ❖ Общая схема реализации артефакта предметно-ориентированного проектирования

Выбор реализации: проект Helidon MP

Первый шаг – выбор реализации MicroProfile, которая будет использоваться для реализации артефактов предметно-ориентированного проектирования. Ранее уже отмечалось, что выбор реализации этой платформы достаточно широк. Для нашей версии мы выбираем проект Helidon MP (https://helidon.io) компании Oracle.

Проект Helidon MP поддерживает спецификации платформы Eclipse MicroProfile, он спроектирован так, чтобы быть простым в использовании и работать на веб-сервере с быстрой ответной реакцией с чрезвычайно малой перегрузкой. В дополнение к полному соответствию спецификациям основного набора и набора поддержки Helidon MP также предоставляет набор расширений, включающих поддержку gRPC, Jedis (поддержка библиотеки Redis), HikariCP (библиотека поддержки пула соединений – Connection Pool) и JTA/JPA.

Общий обзор функциональных возможностей, поддерживаемых проектом Helidon MP, показан на рис. 4.7.

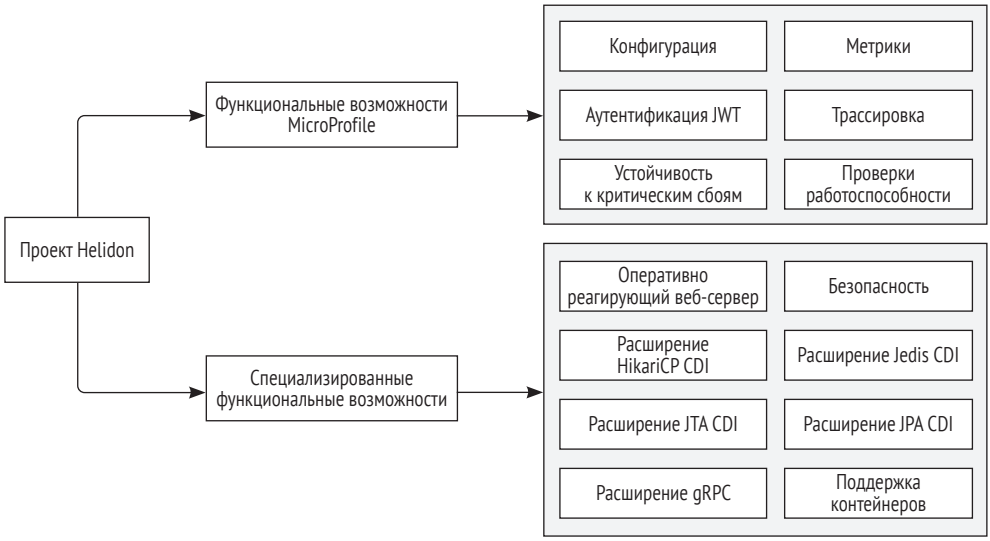

Рис. 4.7 ❖ Схема поддержки проектом Helidon MP спецификаций MicroProfile и дополнительных расширений

Текущая версия проекта Helidon MP 1.2 с поддержкой платформы Eclipse MicroProfile 2.2.

Реализация Cargo Tracker: ограниченные контексты

Этот вариант реализации начинается с разделения монолитного приложения Cargo Tracker на набор микросервисов. Для этого предметная область (домен) Cargo Tracker подразделяется на набор бизнес-функций/поддоменов. На рис. 4.8 показаны следующие бизнес-функции (business capabilities) предметной области (домена) Cargo Tracker:

- заказ груза (Booking) – отвечает за все операции, относящиеся к заказу нового груза, к назначению маршрута доставки груза и к любым обновлениям состояния груза (например, изменение пункта назначения или отмена заказа груза);
- транспортная обработка груза (Handling) – отвечает за все операции, связанные с обработкой груза в различных портах, являющихся частью рейса (маршрута доставки) груза. К ним относятся операции по транспортной обработке грузов или инспектирование груза (например, проверка следования груза по правильному маршруту);
- отслеживание процесса доставки груза (Tracking) – предоставляет интерфейс конечному пользователю (заказчику) для точного отслеживания процесса доставки груза;
- назначение маршрута доставки груза (Routing) – отвечает за все операции, связанные с планированием и сопровождением маршрута доставки груза.

Поддомены в предметно-ориентированном проектировании действуют в так называемой области задачи (или в пространстве задачи – problem space), поэтому и для них необходимо создать решения. В предметно-ориентированном проектировании это достигается с помощью концепции ограниченных контекстов. Проще говоря, ограниченные контексты действуют в области задачи, т. е. ограниченные контексты представляют действительный артефакт решения для конкретной области задачи.

Для рассматриваемой здесь реализации ограниченный контекст моделируется как содержащий единственный микросервис или набор микросервисов. Для этого имеются очевидные причины, поскольку независимость, обеспечиваемая ограниченными контекстами, полностью соответствует основополагающему принципу, требуемому архитектурой на основе микросервисов. Все операции, управляющие состоянием ограниченного контекста (команды для изменения состояния, запросы для извлечения состояния или события для публикации состояния), являются частью микросервисов.

С точки зрения процедуры развертывания, показанной на рис. 4.8, каждый ограниченный контекст представляет собой отдельный самодостаточный элемент развертывания. Элемент развертывания может быть упакован в формате файла fat JAR или как образ контейнера Docker. Поскольку проект Helidon MP предоставляет превосходную поддержку контейнеров Docker, мы выбрали именно этот формат пакетов развертывания.

Микросервисам потребуется хранилище данных (DataStore) для сохранения их состояния. Мы выбрали применение шаблона с отдельной базой данных для каждого сервиса (Database per service), т. е. каждый микросервис будет иметь собственное отдельное хранилище данных. Мы имеем возможность выбора технологии с использованием различных языков и сред программирования для звеньев нашего приложения, и такой же многовариантный выбор предоставлен и для хранилищ данных. Можно выбрать обычную реляционную базу данных (например, Oracle, MySQL, PostgreSQL), базу данных типа NoSQL (например, MongoDB, Cassandra) или даже резидентную в оперативной памяти базу данных (например, Redis). Выбор главным образом зависит от требований к масштабируемости и от типа варианта использования выбираемой базы данных конкретными микросервисами. В рассматриваемом здесь примере реализации в качестве хранилища данных выбрана СУРБД MySQL.

Прежде чем перейти непосредственно к реализации, необходимо сделать краткое отступление о языке, который будет использоваться в дальнейшем. Терминология ограниченного контекста специализирована для предметно-ориентированного проектирования, но поскольку это книга о предметно-ориентированном проектировании, то использование в основном соответствующей терминологии вполне оправдано. Но эта глава посвящена также и реализации микросервисов. Как уже отмечалось ранее, рассматриваемая здесь реализация моделирует ограниченный контекст как микросервис. Поэтому в дальнейшем термин «ограниченный контекст», по существу, означает то же самое, что и термин «микросервис».

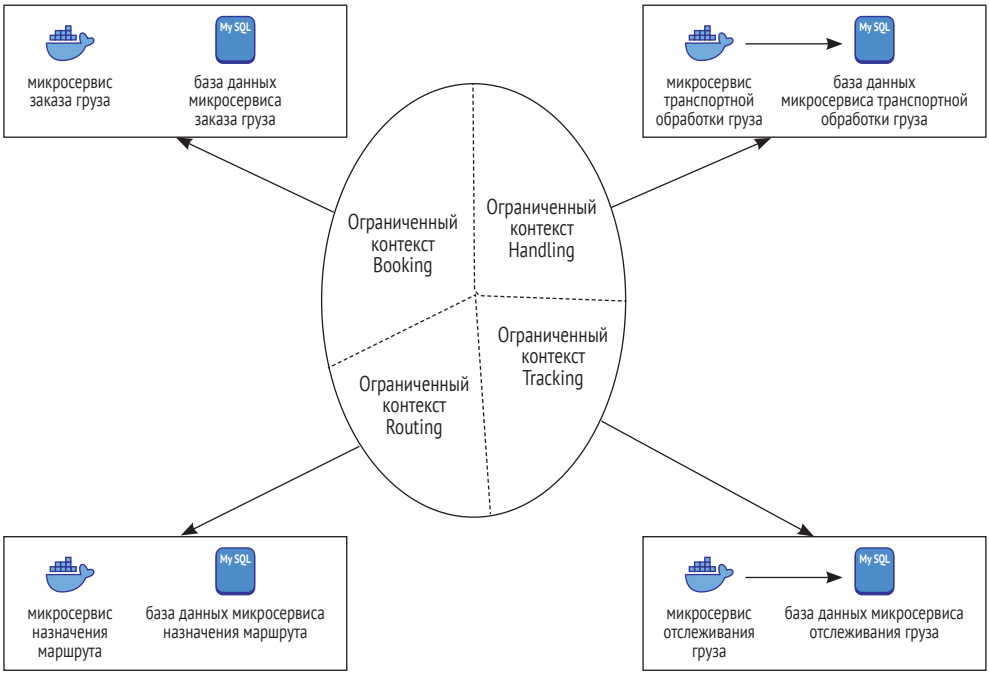

Рис. 4.8 ❖ Артефакты ограниченных контекстов

Ограниченные контексты: создание пакетов

Создание пакетов ограниченных контекстов подразумевает логическое группирование артефактов предметно-ориентированного проектирования в отдельный независимо развертываемый самодостаточный артефакт. Каждый из ограниченных контекстов будет сформирован как приложение Eclipse MicroProfile. Приложения Eclipse MicroProfile самодостаточны в том смысле, что они содержат все зависимости, конфигурации и среду времени выполнения, т. е. они не имеют каких-либо внешних зависимостей (как сервер приложений), необходимых для работы.

Структура приложения Eclipse MicroProfile показана на рис. 4.9.

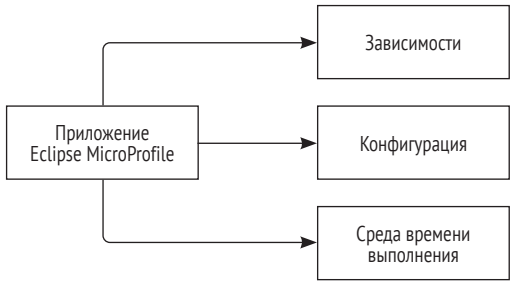

Рис. 4.9 ❖ Структура приложения Eclipse MicroProfile

Проект Helidon MP предлагает прототип Maven (helidon-quickstart-mp) в качестве каркаса приложения Eclipse MicroProfile. В листинге 4.1 показано выполнение команды Helidon MP Maven, используемой для генерации приложения MicroProfile для ограниченного контекста Booking (заказ груза).

Листинг 4.1 ❖ Прототип Helidon MP для быстрой генерации каркаса приложения

```
mvn archetype:generate -DinteractiveMode=false -DarchetypeGroupId=io.helidon.archetypes
    -DarchetypeArtifactId=helidon-quickstart-mp -DarchetypeVersion=1.2
    -DgroupId=com.practicalddd.cargotracker
    -DartifactId=bookingms -Dpackage=com.practicalddd.cargotracker.bookingms
```

Исходный код генерируется в соответствии с каркасом, содержащим класс `Main`. Класс `Main` содержит главный метод `main`, который активизирует веб-сервер Helidon MP при запуске приложения. На рис. 4.10 показана структура исходного кода, сгенерированного шаблоном каркаса helidon-quickstart-mp.

В дополнение к главному классу `Main` также генерируется шаблон файла ресурсов REST (Greeter) для вспомогательного начального тестирования приложения, файл конфигурации MicroProfile (microprofile-config.properties), который можно использовать для настройки конфигурационной информации приложения, а также файл beans.xml для интеграции CDI.

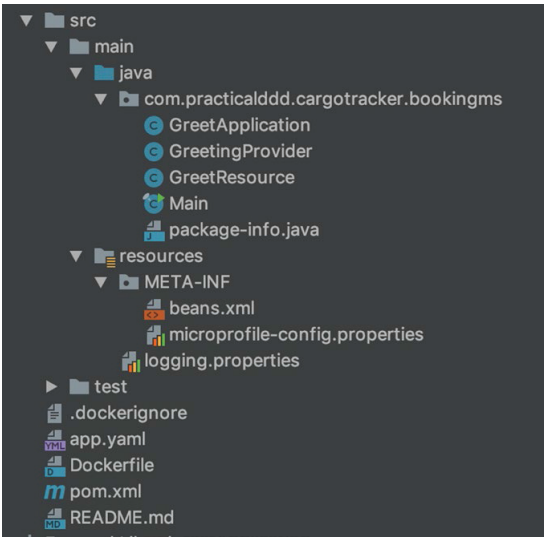

Рис. 4.10 ❖ Проект, сгенерированный с помощью каркаса Helidon

Запустить это приложение можно двумя способами.

- **Как JAR-файл.**
  Результатом сборки этого проекта будет JAR-файл (bookingms.jar). Его запуск как простого JAR-файла с помощью команды `java -jar bookingms.jar` активизирует веб-сервер Helidon MP с определенным в конфигура-

ции портом (8080) и сделает ресурс Greeter REST доступным по адресу `http://<<имя_машины>>:8080/greet`.
Можно воспользоваться утилитой curl для тестирования ресурса Greeter REST с помощью команды:

```
curl -X GET http://<<имя_машины>>:8080/greet
```

В результате выполнения этой команды будет выведено сообщение «Hello World». Это свидетельствует о том, что экземпляр приложения Booking Microservices Helidon MP активен и правильно работает как JAR-файл.

- **Как образ контейнера Docker.**
  Другой способ, предлагаемый средой Helidon MP, – возможность создания и запуска приложения MicroProfile как образа контейнера Docker. При этом сохраняется неизменной основа приложения MicroProfile и предоставляются возможности создания приложений для облачной среды.
  Создание образа контейнера Docker выполняется с помощью следующей команды:

```
docker build -t bookingms
```

Следующая команда предназначена для запуска образа контейнера Docker:

```
docker run -rm -p 8080:8080 bookingms:latest
```

И в этом случае также можно воспользоваться утилитой curl для тестирования ресурса Greeter REST:

```
curl -X GET http://<<имя_машины>>:8080/greet
```

Выводится сообщение «Hello World» как подтверждение того, что экземпляр приложения Booking Microservice Helidon MP активен и правильно работает как образ контейнера Docker.

Так как ранее был выбран подход с использованием контейнеров, микросервисы в приложении Cargo Tracker будут создаваться и запускаться как образы контейнеров Docker.

Ограниченные контексты: структура пакета

После принятия решения о выборе типа пакета следующим шагом является определение структуры пакета для каждого ограниченного контекста, т. е. логическое объединение в группы различных артефактов предметно-ориентированного проектирования MicroProfile в единый независимо развертываемый артефакт. Логическое объединение в группы предполагает определение структуры пакета, в котором размещаются различные артефакты предметно-ориентированного проектирования MicroProfile для получения общего решения для конкретного ограниченного контекста.

Самый верхний уровень структуры пакета для любого ограниченного контекста показан на рис. 4.11.

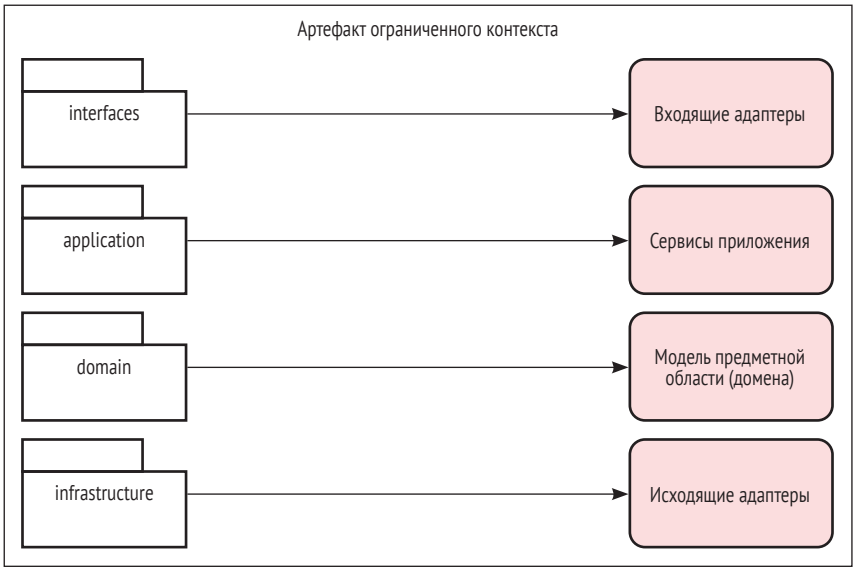

Рис. 4.11 ❖ Структура пакета для ограниченных контекстов

Рассмотрим более подробно структуру пакета для ограниченных контекстов.

Интерфейсы

Этот пакет включает все исходящие интерфейсы для конкретного ограниченного контекста, классифицированные по протоколу обмена данными. Главная цель пакета interfaces – согласование протокола от лица модели предметной области (домена) (например, REST API, WebSocket, FTP, какой-либо специализированный протокол).

В качестве примера рассмотрим предоставляемые ограниченным контекстом Booking (заказ груза) REST API для передачи запросов на изменение состояния, т. е. команд (например, команда заказа груза Book Cargo, команда назначения маршрута для груза Assign Route to Cargo). Кроме того, ограниченный контекст Booking предоставляет REST API для передачи запросов на извлечение состояния, т. е. собственно запросов (Queries) (например, подробная информация о заказанном грузе, список всех грузов). Этот тип запросов объединяется в пакет rest.

Здесь также имеются обработчики событий, которые подписываются на различные события, генерируемые другими ограниченными контекстами. Все обработчики событий объединяются в пакет eventhandlers. В дополнение к этим двум пакетам пакет interfaces также содержит пакет transform, который используется для преобразования входящих данных API-ресурсов/событий в соответствующий формат модели команды/запроса, требуемый моделью предметной области (домена).

Поскольку необходима поддержка REST, событий и операций преобразования данных, определена структура пакета, показанная на рис. 4.12.

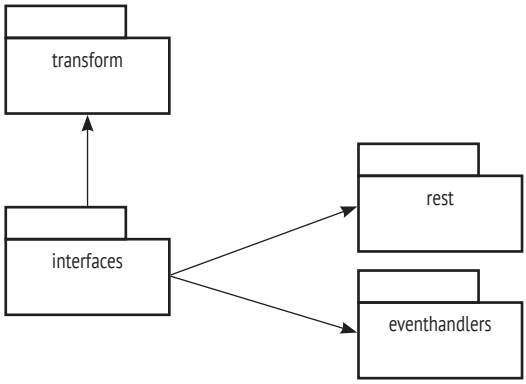

Рис. 4.12 ❖ Структура пакета interfaces

Приложение

Сервисы приложения действуют как внешний элемент модели предметной области (домена) ограниченного контекста. Они предоставляют внешние сервисы для регулирования и направления команд и запросов в модель предметной области (домена) более низкого уровня. В сервисах приложения также формируются исходящие вызовы, направляемые в другие ограниченные контексты, как часть процесса обработки команд и запросов.

Сервисы приложения:

- участвуют в регулировании и направлении команд и запросов;
- при необходимости вызывают компоненты инфраструктуры как часть процесса обработки команд и запросов;
- предоставляют централизованные функции общего назначения (например, ведение журналов, обеспечение безопасности, метрики) для модели предметной области (домена) более низкого уровня;
- выполняют исходящие вызовы, направленные в другие ограниченные контексты.

Структура пакета application показана на рис. 4.13.

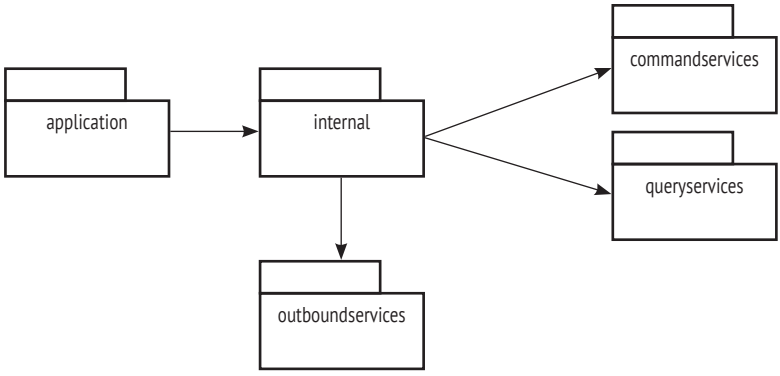

Рис. 4.13 ❖ Структура пакета сервисов приложения

Предметная область (домен)

Этот пакет содержит модель предметной области (домена) ограниченного контекста. Его можно назвать сердцем модели предметной области (домена) ограниченного контекста, содержащим реализацию основной бизнес-логики.

Ниже перечислены основные классы этого ограниченного контекста:

- агрегаты;
- сущности;
- объекты-значения;
- команды;
- события.

Структура пакета domain показана на рис. 4.14.

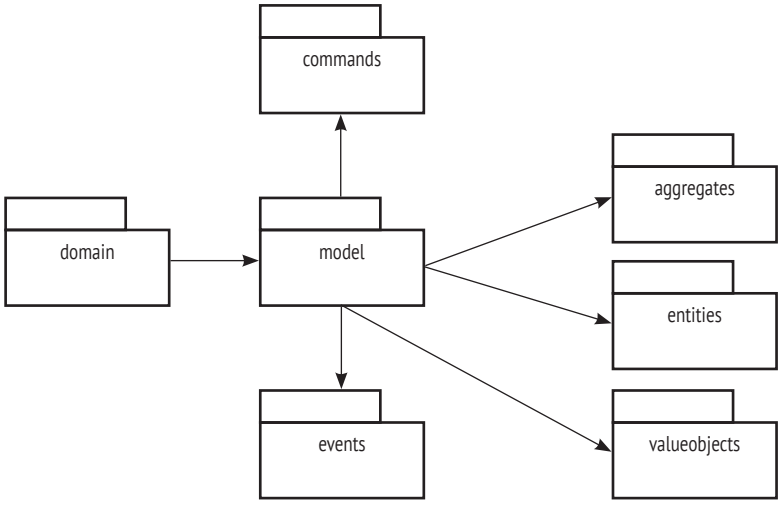

Рис. 4.14 ❖ Структура пакета для модели предметной области

Инфраструктура

Пакет infrastructure предназначен для достижения следующих четырех основных целей:

- когда ограниченный контекст принимает запрос на операцию, связанную с его состоянием (изменением состояния, извлечением состояния), ему необходим нижележащий репозиторий для выполнения этой операции. В рассматриваемом здесь примере репозиторием является экземпляр базы данных MySQL. Пакет инфраструктуры содержит все необходимые компоненты, которые требуются ограниченному контексту для обмена данными с нижележащим репозиторием. Как часть рассматриваемого здесь проекта мы будем использовать JPA или JDBC для реализации этих компонентов;

- когда ограниченный контекст должен передать сообщение о событии изменения состояния, ему необходима нижележащая инфраструктура событий для публикации события изменения состояния. В рассматриваемой здесь реализации будет использоваться брокер сообщений (RabbitMQ) как нижележащая инфраструктура событий. Пакет infrastructure содержит все необходимые компоненты, которые требуются ограниченному контексту для обмена данными с нижележащим брокером сообщений;
- когда ограниченному контексту необходим обмен данными в синхронном режиме с другим ограниченным контекстом, требуется нижележащая инфраструктура для поддержки обмена данными сервиса с сервисом через REST API. Пакет infrastructure содержит все необходимые компоненты, которые требуются ограниченному контексту для обмена данными с другим ограниченным контекстом;
- последний элемент, включенный в инфраструктурный уровень, – любой тип конфигурации, специализированной для MicroProfile.

Структура пакета infrastructure показана на рис. 4.15.

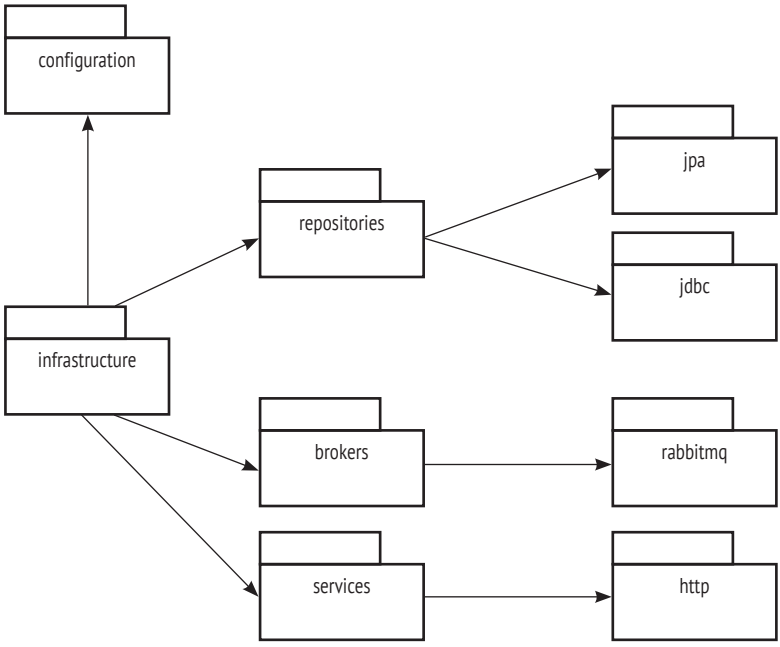

Рис. 4.15 ❖ Структура пакета, содержащего компоненты инфраструктуры

Полная обобщенная схема структуры всего пакета в целом для любого ограниченного контекста показана на рис. 4.16.

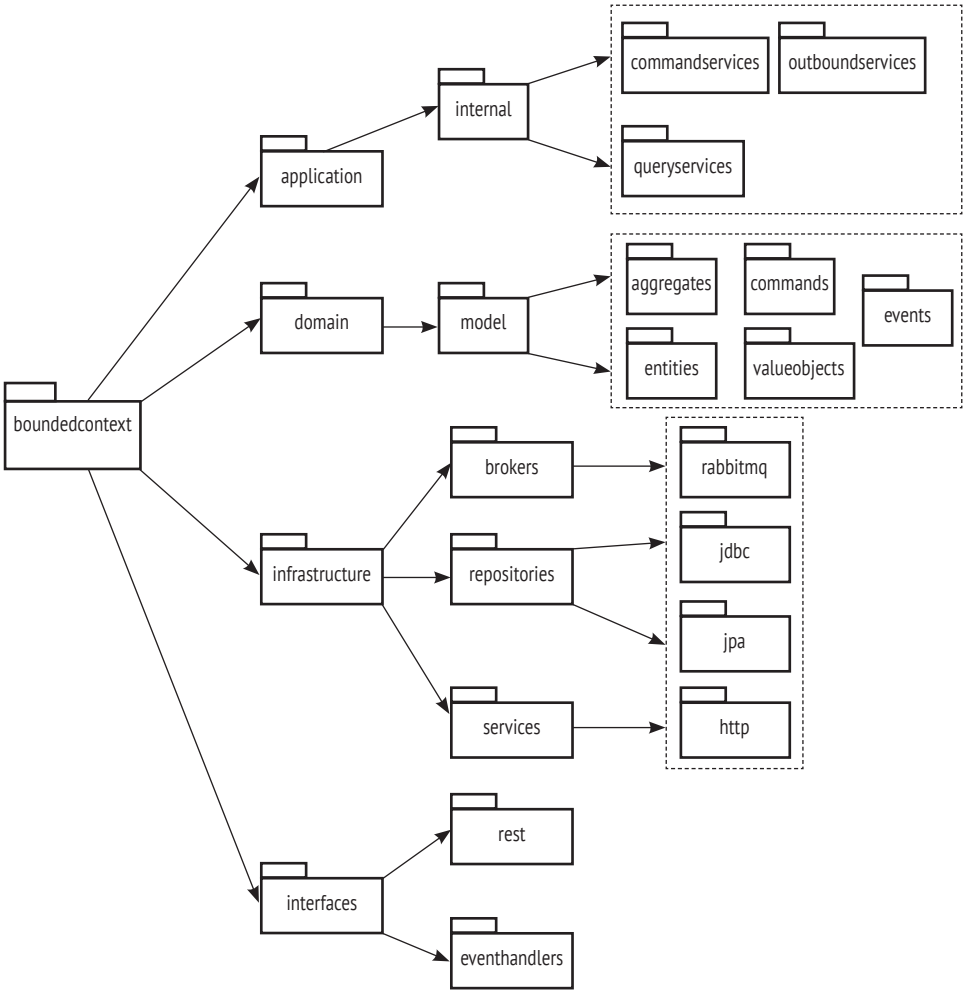

Рис. 4.16 ❖ Общая структура пакета для любого ограниченного контекста

На этом завершается реализация ограниченных контекстов для рассматриваемого здесь приложения Cargo Tracker на основе микросервисов. Каждый из ограниченных контекстов реализован как приложение MicroProfile с использованием проекта Helidon MP, а в качестве артефакта принят образ контейнера Docker. Ограниченные контексты аккуратно разделены в структуре пакета по модулям с четко разграниченными функциями.

Теперь можно перейти к следующему этапу реализации приложения Cargo Tracker.

Реализация приложения Cargo Tracker

В этом разделе подробно рассматривается реализация Cargo Tracker как приложения на основе микросервисов с использованием предметно-ориентиро-

ванного проектирования и программной среды Eclipse MicroProfile (проект Helidon MP).

Обобщенный обзор логического группирования различных артефактов предметно-ориентированного проектирования показан на рис. 4.17. Здесь можно видеть, что необходима реализация следующих двух групп артефактов:

- модель предметной области (домена), которая будет содержать ранее определенный основной домен/бизнес-логику;
- сервисы предметной области (домена), содержащие сервисы поддержки для рассматриваемой здесь модели основного домена.

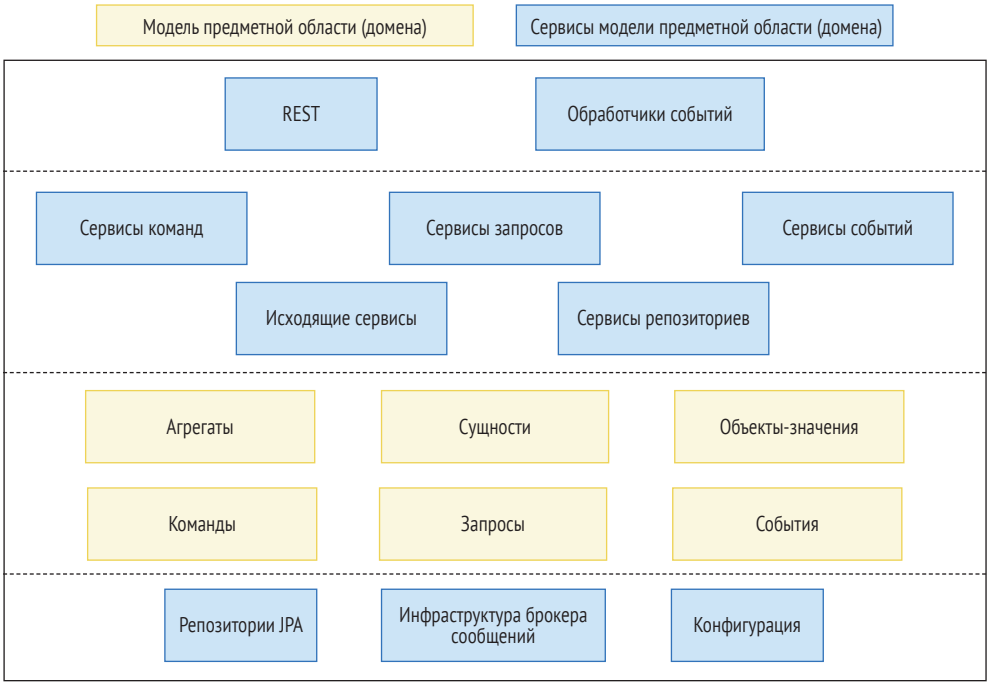

Рис. 4.17 ❖ Логическое группирование артефактов предметно-ориентированного проектирования

В терминах действительной реализации модели предметной области (домена) эти артефакты преобразуются в различные объекты-значения, команды и запросы конкретного ограниченного контекста/микросервисов.

В терминах действительной реализации сервисов модели предметной области (домена) эти артефакты преобразуются в интерфейсы, сервисы приложения и инфраструктуру, которые требует модель предметной области (домена) ограниченного контекста/микросервисов.

Вернемся к приложению Cargo Tracker. На рис. 4.18 показано решение с использованием микросервисов в терминах различных ограниченных контекстов и поддерживаемых ими операций. Схема решения содержит различные команды, которые будет обрабатывать каждый ограниченный контекст, запросы, которые будет обслуживать каждый ограниченный контекст, и собы-

тия, которые будет публиковать и на которые будет подписываться каждый ограниченный контекст. Каждый микросервис представляет собой отдельный независимо развертываемый артефакт с собственным хранилищем данных.

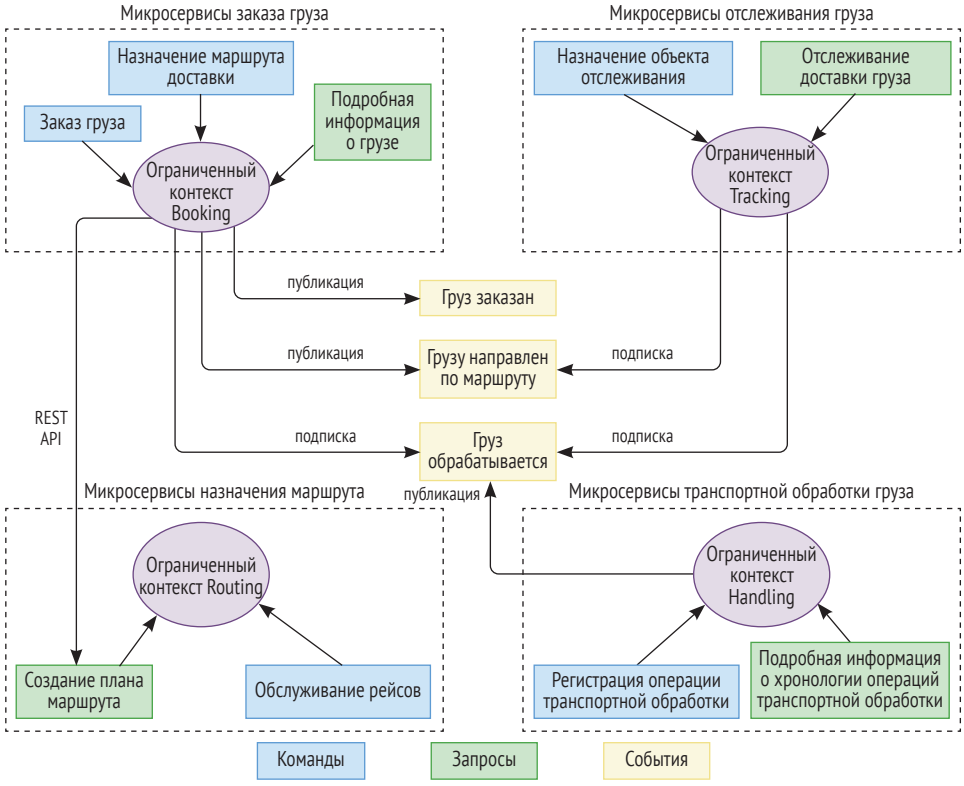

Рис. 4.18 ❖ Решение на основе микросервисов для приложения Cargo Tracker

ПРИМЕЧАНИЕ Некоторые примеры исходного кода реализации могут содержать только фрагменты или сокращенные варианты, помогающие понять общие концепции реализации. Исходный код для текущей главы содержит полную реализацию этих концепций.

Модель предметной области (домена): реализация

Принятая модель предметной области (домена) является центральным функциональным элементом рассматриваемого ограниченного контекста и, как было отмечено выше, имеет набор артефактов, связанных с этой моделью. Реализация этих артефактов выполняется с помощью инструментальных средств, предоставляемых Eclipse MicroProfile.

Ниже перечислены артефакты модели предметной области (домена), которые необходимо реализовать:

- модель основного домена – агрегаты, сущности и объекты-значения;
- операции модели предметной области (домена) – команды, запросы и события.

Рассмотрим подробнее каждый из этих артефактов и определим, какие инструментальные средства, предоставляемые Eclipse MicroProfile, подходят для их реализации.

Модель основного домена: реализация

Реализация основной предметной области (домена) для любого ограниченного контекста включает идентификацию тех артефактов, которые ясно выражают бизнес-цели этого ограниченного контекста. На самом верхнем уровне это включает идентификацию и реализацию агрегатов, сущностей и объектов-значений.

Агрегаты, сущности и объекты-значения

Агрегаты являются главными элементами модели предметной области (домена). Напомним, что мы определили четыре агрегата, по одному в каждом ограниченном контексте, как показано на рис. 4.19.

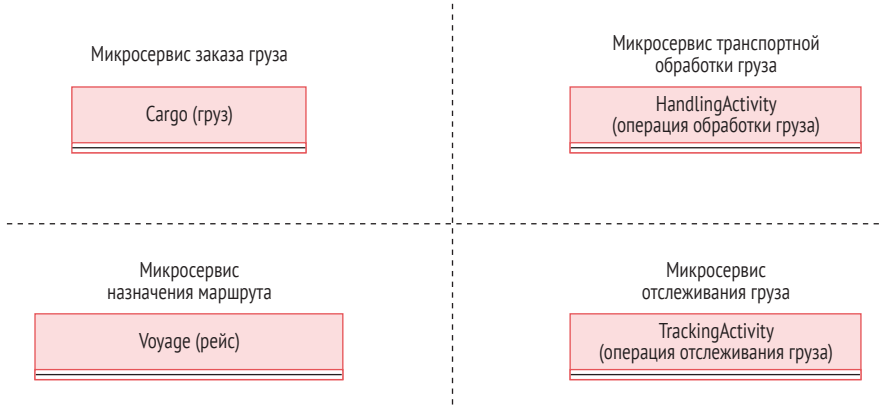

Рис. 4.19 ❖ Агрегаты в ограниченных контекстах/микросервисах

Реализация агрегата включает следующие этапы:

- реализацию класса агрегата;
- обеспечение полноценности предметной области (домена) с помощью бизнес-атрибутов;
- реализацию сущностей и объектов-значений.

Реализация класса агрегата

Поскольку выбрано использование СУРБД MySQL в качестве хранилища данных для каждого ограниченного контекста, будет применяться интерфейс JPA (Java Persistent API) по соответствующей спецификации Java EE. JPA предоставляет стандартный способ определения и реализации сущностей и объектов-значений, которые взаимодействуют с хранилищами данных MySQL более низкого уровня.

Интеграция JPA: Helidon MP

Проект Helidon MP обеспечивает поддержку использования JPA через внешний механизм интеграции. Для включения этой поддержки в приложение необходимо добавить некоторые дополнительные параметры конфигурации и зависимости:

`pom.xml`

В листинге 4.2 показаны изменения, которые необходимо внести в файл зависимостей *pom.xml*, генерируемый Helidon MP.

Список зависимостей содержит следующие пункты:

- поддержку интеграции JPA Helidon MP (helidon-integrations-cdi-jpa);
- использование HikariCP для механизма сопровождения пула соединений с источником данных;
- драйвер библиотеки MySQL для Java (mysql-connector-java).

Листинг 4.2 ❖ Зависимости в файле pom.xml

```
<dependency>
    <groupId>io.helidon.integrations.cdi</groupId>
    <artifactId>helidon-integrations-cdi-datasource-hikaricp</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
    <groupId>io.helidon.integrations.cdi</groupId>
    <artifactId>helidon-integrations-cdi-jpa</artifactId>
</dependency>
<dependency>
    <groupId>io.helidon.integrations.cdi</groupId>
    <artifactId>helidon-integrations-cdi-jpa-weld</artifactId>
</dependency>
<dependency>
    <groupId>io.helidon.integrations.cdi</groupId>
    <artifactId>helidon-integrations-cdi-eclipselink</artifactId>
</dependency>
<dependency>
    <groupId>jakarta.persistence</groupId>
    <artifactId>jakarta.persistence-api</artifactId>
</dependency>
<dependency>
    <groupId>io.helidon.integrations.cdi</groupId>
    <artifactId>helidon-integrations-cdi-jta</artifactId>
</dependency>
<dependency>
    <groupId>io.helidon.integrations.cdi</groupId>
    <artifactId>helidon-integrations-cdi-jta-weld</artifactId>
</dependency>
<dependency>
    <groupId>javax.transaction</groupId>
    <artifactId>javax.transaction-api</artifactId>
```

```
</dependency>
microprofile-config
```

Необходимо определить конфигурацию свойств соединения для каждого экземпляра базы данных MySQL. В листинге 4.3 показана конфигурация свойств, которые должны быть добавлены. Возможно, потребуется замена значений в соответствии с вашей конкретной конфигурацией экземпляров MySQL, если это будет необходимо.

Листинг 4.3 ❖ Информация о конфигурации для соединения с источником данных

```
javax.sql.DataSource.<<имя_ограниченного_контекста>>.dataSourceClassName=com.mysql.
cj.jdbc.MysqlDataSource
javax.sql.DataSource.<<имя_ограниченного_контекста>>.dataSource.
url=jdbc:mysql://<<имя_машины>>:<<номер_порта_машины>>/<<имя_экземпляра_БД_MySQL>>
javax.sql.DataSource.<<имя_ограниченного_контекста>>.dataSource.user=
<<UserID_экземпляра_БД_MySQL>>
javax.sql.DataSource.<<имя_ограниченного_контекста>>.dataSource.password=
<<пароль_для_экземпляра_БД_MySQL>>
persistence.xml
```

Завершающим шагом является конфигурирование JPA «persistence-unit», отображаемого в источник данных. Конфигурация определяется в файле *microprofile-config*, как показано в листинге 4.4.

Листинг 4.4 ❖ Информация о конфигурации элемента персистентности

```
<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="<<имя_ограниченного_контекста>>" transaction-type="JTA">
    <jta-data-source><<источник_данных_ограниченного_контекста>></jta-data-source>
</persistence-unit>
</persistence>
```

Теперь все готово к реализации JPA в приложениях MicroProfile. Все создаваемые агрегаты во всех ранее определенных ограниченных контекстах реализуют один и тот же механизм, если не указан особый специализированный вариант.

Каждый класс корневого агрегата реализуется как объект JPA. JPA не предоставляет специальные аннотации для обозначения конкретного класса как корневого агрегата, поэтому принимается обычное обозначение старого объекта Java (POJO) и используется стандартная аннотация JPA `@Entity`. Рассмотрим пример ограниченного контекста Booking (заказ груза) с корневым агрегатом Cargo. В листинге 4.5 показан небольшой фрагмент кода, необходимый для объекта JPA.

Листинг 4.5 ❖ Корневой агрегат Cargo, реализованный как объект JPA

```
package com.practicalddd.cargotracker.bookingms.domain.model.aggregates;
import javax.persistence.*;
@Entity // Маркер объекта JPA
public class Cargo {
}
```

Для каждого объекта JPA требуется идентификатор. Для реализации идентификатора агрегата выбран технический идентификатор (первичный ключ) агрегата Cargo, выведенный из последовательности MySQL. В дополнение к техническому идентификатору также определяется бизнес-ключ (Business Key).

Бизнес-ключ явно выражает бизнес-цель идентификатора агрегата, т. е. идентификатор заказа нового груза, следовательно, это ключ, предъявляемый внешним потребителям (пользователям) модели предметной области (подробности об этом будут изложены несколько позже). С другой стороны, технический ключ – это исключительно внутреннее представление идентификатора агрегата, удобное для обслуживания отношений в ограниченном контексте между агрегатами и зависящими от них объектами (см. сущности и объекты-значения в следующих подразделах).

Продолжая рассматривать пример агрегата Cargo в ограниченном контексте Booking, мы прямо сейчас добавляем технический ключ и бизнес-ключ в реализацию класса агрегата.

Это показано в листинге 4.6. Аннотация `@Id` определяет первичный ключ агрегата Cargo. Для обозначения бизнес-ключа нет специализированной аннотации, поэтому он реализован как обычный старый объект Java (POJO) `BookingId` и включен в состав агрегата с помощью аннотации `@Embedded`, предоставляемой JPA.

Листинг 4.6 ❖ Реализация идентификатора корневого агрегата Cargo

```
package com.practicalddd.cargotracker.bookingms.domain.model.aggregates;
import javax.persistence.*;
@Entity
public class Cargo {
  @Id // Аннотация идентификатора, предоставляемая JPA.
  // На основе последовательности, сгенерированной MySQL.
  @GeneratedValue( strategy = GenerationType.AUTO )
  private Long id;
  @Embedded // Аннотация, позволяющая использовать бизнес-объекты вместо простейших типов.
  private BookingId bookingId; // Бизнес-идентификатор.
}
```

В листинге 4.7 показана реализация класса бизнес-ключа `BookingId`.

Листинг 4.7 ❖ Реализация класса бизнес-ключа BookingId

```
package com.practicalddd.cargotracker.bookingms.domain.model.aggregates;
import javax.persistence.*;
import java.io.Serializable;
/**
 * Идентификатор бизнес-ключ для агрегата Cargo.
 */
@Embeddable
public class BookingId implements Serializable {
  @Column( name="booking_id" )
  private String bookingId;
  public BookingId() {}
```

```
  public BookingId( String bookingId ) { this.bookingId = bookingId; }
  public String getBookingId() { return this.bookingId; }
}
```

Теперь у нас есть основной каркас реализации агрегата (в данном случае агрегата Cargo) с использованием JPA. Для других агрегатов используется тот же механизм реализации за исключением ограниченного контекста Handling Activity (операция транспортной обработки груза). Так как это журнал событий, принято решение реализовать только один ключ для этого агрегата, т. е. `ActivityId` (идентификатор операции).

На рис. 4.20 показана общая схема первоначальной реализации всех агрегатов.

Рис. 4.20 ❖ Первоначальная реализация агрегатов

Обеспечение полноценности предметной области: бизнес-атрибуты

Подготовив первоначальные реализации агрегатов, можно переходить к следующему этапу – обеспечению полноценности предметной области (домена). Агрегат любого ограниченного контекста должен обладать способностью выражения бизнес-языка своего ограниченного контекста. По существу, в точной технической терминологии это означает, что агрегат не должен быть неполноценным (anemic), т. е. не должен содержать только методы установки и считывания значений (getter/setter methods).

Неполноценный агрегат (anemic aggregate) противоречит основному принципу предметно-ориентированного проектирования, поскольку, в сущности, это означало бы, что бизнес-язык выражается на нескольких уровнях приложения. В свою очередь это ведет к тому, что некоторая часть программного обеспечения становится чрезвычайно неудобной в эксплуатации и сопровождении при долговременном использовании.

Но как реализовать агрегат полноценной предметной области (домена)? Краткий ответ: с помощью бизнес-атрибутов и бизнес-методов. В этом подразделе мы сосредоточимся на бизнес-атрибутах, а бизнес-методы будут рассматриваться как часть реализации операции модели предметной области (домена).

Бизнес-атрибуты агрегата содержат состояние агрегата в форме атрибутов, описываемых с использованием бизнес-терминов, а не технической терминологии.

Рассмотрим подробнее бизнес-атрибуты на примере агрегата Cargo.

После преобразования состояния в бизнес-концепции агрегат Cargo имеет следующие атрибуты:

- исходную локацию (Origin Location) груза;
- количество заказанного (Booking Amount) груза;
- спецификацию маршрута (Route specification) (исходную локацию, пункт назначения, конечный срок прибытия в пункт назначения);
- план маршрута доставки (Itinerary), который назначается для груза на основе спецификации маршрута. План маршрута доставки состоит из нескольких этапов (Legs), по которым груз может перемещаться, чтобы дойти до пункта назначения;
- ход процесса доставки (Delivery Progress) груза присваивается вместе со спецификацией и планом маршрута доставки. Ход процесса доставки предоставляет подробную информацию о состоянии маршрута, состоянии транспорта, текущем рейсе доставки груза, последней известной локации груза, следующей запланированной операции и последней операции, которая производилась с грузом.

На рис. 4.21 показан агрегат Cargo и его отношения с зависящими от него объектами. Следует особо отметить возможность явного представления агрегата Cargo исключительно в бизнес-терминах.

JPA предоставляет набор аннотаций (`@Embedded`, `@Embeddable`), которые помогают реализовать класс этого агрегата с использованием бизнес-объектов.

В листинге 4.8 показан пример агрегата Cargo со всеми его зависимостями, смоделированными как бизнес-объекты.

Листинг 4.8 ❖ Зависимые бизнес-объекты для корневого агрегата Cargo

```
package com.practicalddd.cargotracker.bookingms.domain.model.aggregates;
import javax.persistence.*;

import com.practicalddd.cargotracker.bookingms.domain.model.entities.*;
import com.practicalddd.cargotracker.bookingms.domain.model.valueobjects.*;
@Entity
public class Cargo {
  @Id
  @GeneratedValue( strategy = GenerationType.AUTO )
  private Long id;
  @Embedded
  private BookingId bookingId; // Идентификатор агрегата.
  @Embedded
  private BookingAmount bookingAmount; // Количество заказанного груза.
  @Embedded
  private Location origin; // Исходная локация груза.
  @Embedded
  private RouteSpecification routeSpecification; // Спецификация маршрута доставки груза.
  @Embedded
  private CargoItinerary itinerary; // План маршрута доставки груза.
```

```
  @Embedded
  private Delivery delivery; // Проверки хода процесса доставки груза в соответствии с
                             // реальной спецификацией и планом маршрута доставки груза.
}
```

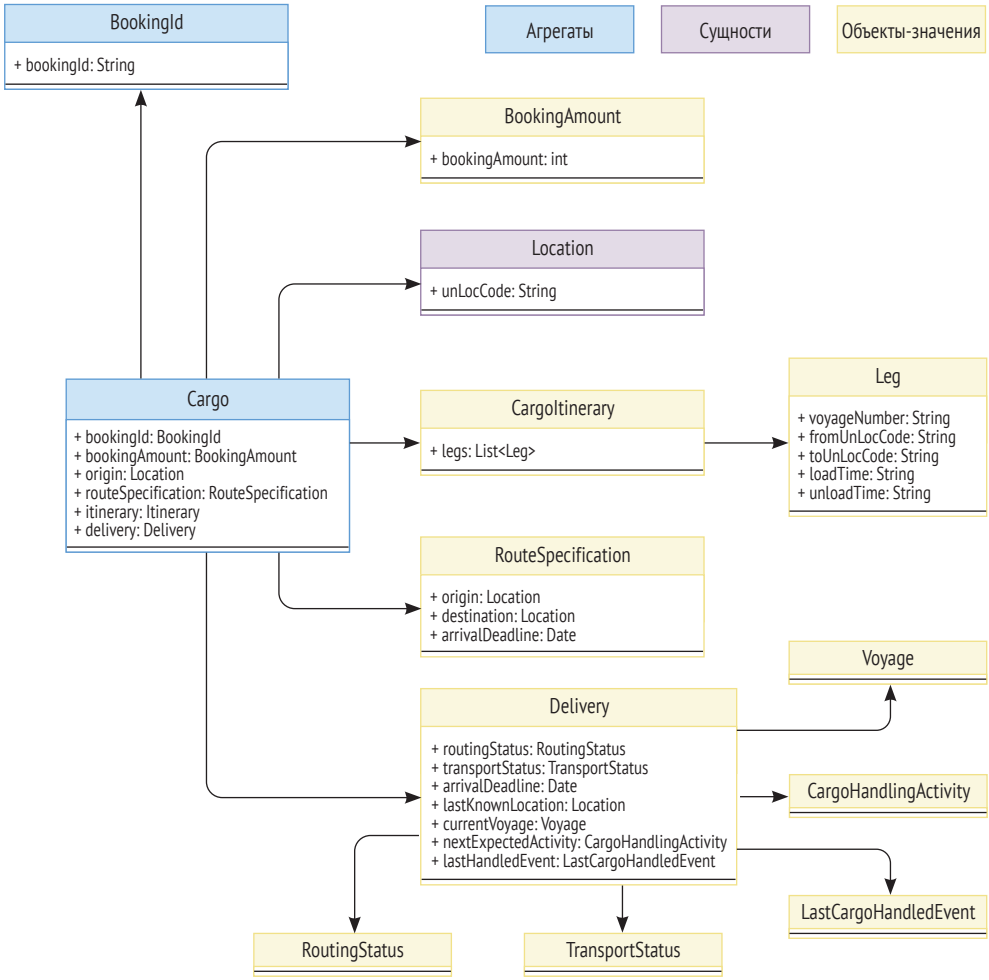

Рис. 4.21 ❖ Агрегат Cargo и зависимые от него объекты

Зависимые классы для агрегата моделируются либо как объекты-сущности, либо как объекты-значения. Напомним, что объекты-сущности в ограниченном контексте обладают собственной идентичностью, но всегда существуют только в корневом агрегате, т. е. не могут существовать независимо. Кроме того, объекты-сущности никогда не изменяются на протяжении всего жизненного цикла агрегата. С другой стороны, объекты-значения не обладают собственной идентичностью и легко заменяются в любом экземпляре агрегата.

Продолжая рассматривать пример агрегата Cargo, получаем следующие объекты:

- исходную локацию (Origin) груза как объект-сущность (Location). Этот объект не может измениться в любом экземпляре агрегата Cargo, следовательно, моделируется как объект-сущность;
- количество заказанного груза (Booking Amount), спецификацию маршрута доставки (Route Specification), план маршрута доставки (Cargo Itinerary), назначенный для конкретного груза, и процесс доставки (Delivery of the Cargo) как объекты-значения. Эти объекты являются заменяемыми в любом экземпляре агрегата Cargo, следовательно, моделируются как объекты-значения.

Рассмотрим подробнее конкретные ситуации (сценарии) и обоснуем, почему эти объекты определены как объекты-значения, так как это весьма важное решение при моделировании предметной области (домена):

- при новом заказе груза создается новая спецификация маршрута доставки и пустой план маршрута доставки, но отсутствует ход процесса доставки;
- после назначения плана маршрута доставки груза пустой план маршрута доставки заменяется назначенным планом маршрута доставки;
- по мере перемещения груза по нескольким портам, являющимся частью плана маршрута доставки, ход процесса доставки груза обновляется и заменяется в корневом агрегате;
- если заказчик изменяет пункт назначения груза или предельный срок доставки, то изменяется спецификация маршрута, назначается новый план маршрута доставки, пересчитывается процесс доставки (Delivery), а также изменяется количество заказанного груза.

Все упомянутые выше объекты являются заменяемыми, следовательно, моделируются как объекты-значения. Это простое практическое правило моделирования сущностей и объектов-значений в агрегате.

Реализация объектов-сущностей и объектов-значений

Объекты-сущности и объекты-значения реализуются как встраиваемые (embeddable) объекты JPA с использованием доступной аннотации `@Embeddable`. Затем они включаются в агрегат с использованием аннотации `@Embedded`.

В листингах 4.9 и 4.10 показан механизм включения объектов-сущностей и объектов-значений в класс агрегата.

Рассмотрим подробнее реализацию объектов-сущностей и объектов-значений агрегата Cargo.

В листинге 4.9 показана реализация объекта-сущности `Location`. Обратите внимание на имя пакета, в котором собраны сущности (`model.entities`).

Листинг 4.9 ❖ Реализация класса сущности Location

```
package com.practicalddd.cargotracker.bookingms.domain.model.entities;
import javax.persistence.Column;
import javax.persistence.Embeddable;
/**
 * Класс сущности Location, представленный уникальным 5-значным кодом локации ООН.
 */
@Embeddable
public class Location {
```

```
  @Column( name = "origin_id" )
  private String unLocCode;
  public Location() {}
  public Location( String unLocCode ) { this.unLocCode = unLocCode; }
  public void setUnLocCode( String unLocCode ) { this.unLocCode = unLocCode; }
  public String getUnLocCode() { return this.unLocCode; }
}
```

В листинге 4.10 показан пример реализации объекта-значения `BookingAmount`. Обратите внимание на имя пакета, в котором собраны объекты-значения (`model.valueobjects`).

Листинг 4.10 ❖ Реализация объекта-значения BookingAmount

```
package com.practicalddd.cargotracker.bookingms.domain.model.valueobjects;
import javax.persistence.Column;
import javax.persistence.Embeddable;
/**
 * Представление в модели домена объекта-значения BookingAmount для нового груза.
 * Содержит количество заказанного груза.
 */
@Embeddable
public class BookingAmount {
  @Column( name = "booking_amount", unique = true, updatable= false )
  private Integer bookingAmount;
  public BookingAmount() {}
  public BookingAmount( Integer bookingAmount ) { this.bookingAmount = bookingAmount; }
  public void setBookingAmount( Integer bookingAmount )
              { this.bookingAmount = bookingAmount; }
  public Integer getBookingAmount() { return this.bookingAmount; }
}
```

В листинге 4.11 показан пример реализации объекта-значения `RouteSpecification`.

Листинг 4.11 ❖ Реализация объекта-значения RouteSpecification

```
package com.practicalddd.cargotracker.bookingms.domain.model.valueobjects;
import com.practicalddd.cargotracker.bookingms.domain.model.entities.Location;
import javax.persistence.*;
import javax.validation.constraints.NotNull;
import java.util.Date;
/**
 * Спецификация маршрута доставки заказанного груза.
 */
@Embeddable
public class RouteSpecification {
  private static final long serialVersionUID = 1L;
  @Embedded
  @AttributeOverride( name = "unLocCode", column = @Column( name = "spec_origin_id" ) )
  private Location origin;
  @Embedded
  @AttributeOverride( name = "unLocCode", column = @Column( name = "spec_destination_id" )
)
```

```
  private Location destination;
  @Temporal( TemporalType.DATE )
  @Column( name = "spec_arrival_deadline" )
  @NotNull
  private Date arrivalDeadline;
  public RouteSpecification() {}
  /**
   * @param origin origin location
   * @param destination destination location
   * @param arrivalDeadline arrival deadline
   */
  public RouteSpecification( Location origin, Location destination,
                             Date arrivalDeadline ) {
    this.origin = origin;
    this.destination = destination;
    this.arrivalDeadline = (Date)arrivalDeadline.clone();
  }
  public Location getOrigin() {
    return origin;
  }
  public Location getDestination() {
    return destination;
  }
  public Date getArrivalDeadline() {
    return new Date(arrivalDeadline.getTime());
  }
}
```

Все прочие объекты-значения (CargoItinerary и Delivery) реализуются аналогичным образом с использованием аннотации `@Embeddable` и включаются в агрегат Cargo с помощью аннотации `@Embedded`.

> **ПРИМЕЧАНИЕ** Полная реализация содержится в файлах исходного кода для этой главы.

На рис. 4.22, 4.23 и 4.24 показаны упрощенные диаграммы классов для других агрегатов (HandlingActivity, Voyage, TrackingActivity).

На этом завершается рассмотрение реализации модели основной предметной области (домена). В следующем разделе будет рассматриваться реализация операций модели предметной области (домена).

> **ПРИМЕЧАНИЕ** Исходный код, прилагаемый к этой книге, содержит реализацию модели основной предметной области (домена), демонстрируемую с использованием разделения пакетов. Чтобы получить более подробное представление о типах объектов в модели предметной области (домена), вы можете ознакомиться с исходным кодом в репозитории github.com/practicalddd.

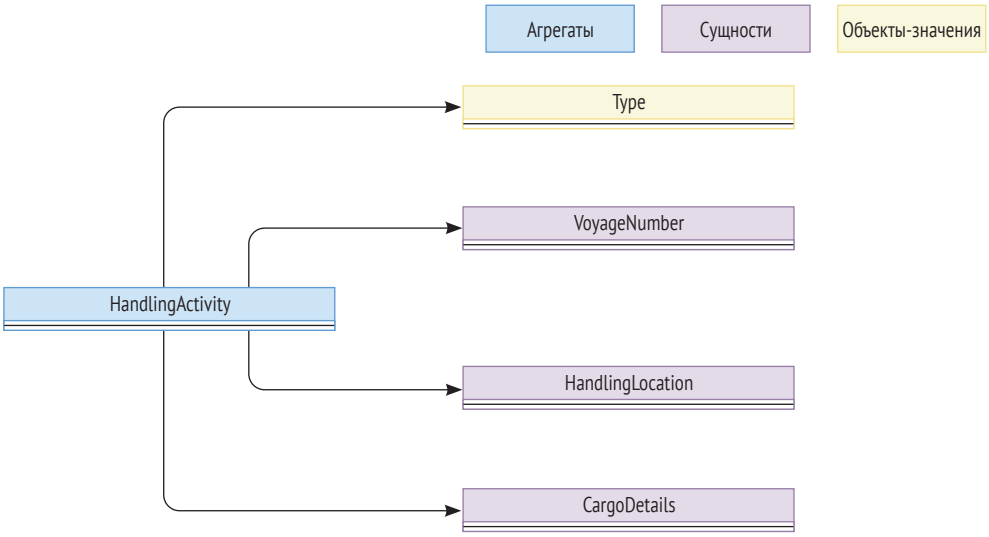

Рис. 4.22 ❖ Класс HandlingActivity и зависимые от него объекты

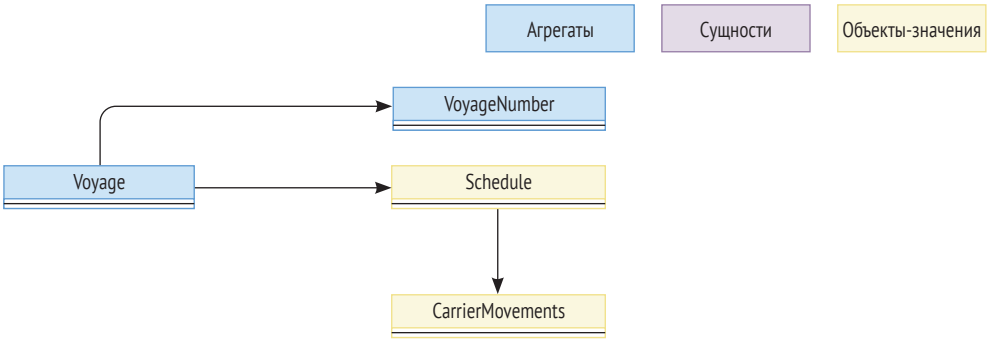

Рис. 4.23 ❖ Класс Voyage и зависимые от него объекты

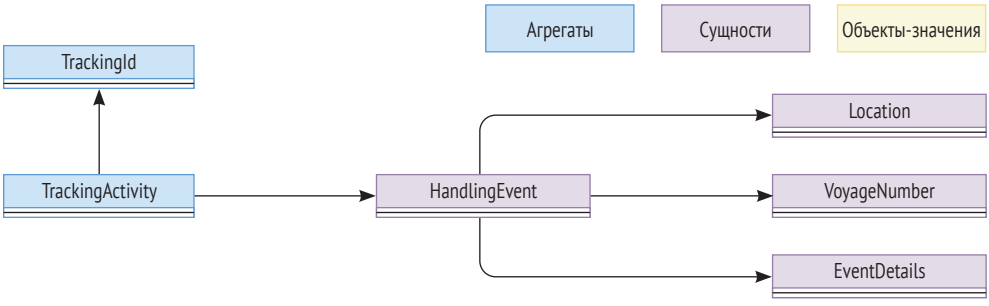

Рис. 4.24 ❖ Класс TrackingActivity и зависимые от него объекты

Операции модели предметной области (домена)

Операции модели предметной области (домена) в ограниченном контексте выполняют любой тип действий, связанных с состоянием агрегата в этом ограниченном контексте. Это операции, изменяющие состояние агрегата (команды), извлекающие текущее состояние агрегата (запросы) или оповещающие об изменении состояния агрегата (события).

Команды

Команды отвечают за изменение состояния агрегата в ограниченном контексте.

Реализация команд в ограниченном контексте подразумевает выполнение следующих шагов:

- идентификации/реализации команд;
- идентификации/реализации обработчиков команд для обработки команд.

Идентификация команд

Идентификация команд охватывает любую операцию, которая влияет на состояние агрегата. Например, ограниченный контекст команд заказа груза содержит следующие операции или команды:

- заказ груза;
- назначение маршрута доставки груза.

Обе эти операции в конечном итоге изменяют состояние агрегата Cargo в соответствующем ограниченном контексте, следовательно, идентифицируются как команды.

Реализация команд

Реализация идентифицированных команд на платформе MicroProfile выполняется с использованием обычных старых объектов Java (POJO). В листинге 4.12 показана реализация класса `BookCargoCommand` для команды заказа груза.

Листинг 4.12 ❖ Реализация класса BookCargoCommand

```
package com.practicalddd.cargotracker.bookingms.domain.model.commands;
import java.util.Date;
/**
 * Класс команды заказа груза.
 */
public class BookCargoCommand {
  private String bookingId;
  private int bookingAmount;
  private String originLocation;
  private String destLocation;
  private Date destArrivalDeadline;
  public BookCargoCommand() {}
  public BookCargoCommand( int bookingAmount, String originLocation, String destLocation,
                           Date destArrivalDeadline ) {
    this.bookingAmount = bookingAmount;
    this.originLocation = originLocation;
```

```
    this.destLocation = destLocation;
    this.destArrivalDeadline = destArrivalDeadline;
  }
  public void setBookingId( String bookingId ) { this.bookingId = bookingId; }
  public String getBookingId() { return this.bookingId; }
  public void setBookingAmount( int bookingAmount ) {
    this.bookingAmount = bookingAmount;
  }
  public int getBookingAmount() {
    return this.bookingAmount;
  }
  public String getOriginLocation() { return originLocation; }
  public void setOriginLocation( String originLocation )
              { this.originLocation = originLocation; }
  public String getDestLocation() { return destLocation; }
  public void setDestLocation( String destLocation ) { this.destLocation = destLocation; }
  public Date getDestArrivalDeadline() { return destArrivalDeadline; }
  public void setDestArrivalDeadline( Date destArrivalDeadline )
              { this.destArrivalDeadline = destArrivalDeadline; }
}
```

Идентификация обработчиков команд

Для каждой команды будет предусмотрен соответствующий обработчик команды (Command Handler). Цель обработчика команды – обработать введенную команду и установить заданное состояние агрегата. Обработчики команд – это единственная часть модели предметной области (домена), где устанавливается состояние агрегата. Это строгое правило, которое необходимо соблюдать, чтобы правильно реализовать полноценную модель предметной области (домена).

Реализация обработчиков команд

Так как платформа Eclipse MicroProfile не предоставляет готовые к прямому использованию функциональные возможности для реализации обработчиков команд, методикой реализации в рассматриваемом примере будет простая идентификация подпрограмм в агрегатах. Эти подпрограммы можно обозначить как обработчики команд (Command Handlers). Для первой (по порядку выполнения) команды заказа груза (Book Cargo) конструктор соответствующего агрегата идентифицируется как обработчик этой команды. Для второй команды назначения маршрута доставки груза (Route Cargo) создается новая подпрограмма `assignToRoute()` как обработчик этой команды.

В листинге 4.13 показан фрагмент кода конструктора агрегата Cargo. Конструктор принимает команду `BookCargoCommand` как входной параметр и устанавливает соответствующее состояние этого агрегата.

Листинг 4.13 ❖ Обработчик команды BookCargoCommand

```
/**
 * Конструктор как обработчик команды нового заказа груза.
 */
public Cargo( BookCargoCommand bookCargoCommand ) {
  this.bookingId = new BookingId( bookCargoCommand.getBookingId() );
```

```
  this.routeSpecification = new RouteSpecification(
                  new Location( bookCargoCommand.getOriginLocation() ),
                  new Location( bookCargoCommand.getDestLocation() ),
                  bookCargoCommand.getDestArrivalDeadline() );
  this.origin = routeSpecification.getOrigin();
  this.itinerary = CargoItinerary.EMPTY_ITINERARY; // Пустой план маршрута доставки,
                                                   // так как грузу еще не назначен маршрут.
  this.bookingAmount = bookingAmount;
  this.delivery = Delivery.derivedFrom( this.routeSpecification, this.itinerary,
                                        LastCargoHandledEvent.EMPTY );
}
```

В листинге 4.14 показан фрагмент кода для обработчика команды `assignToRoute()`. В качестве входного параметра он принимает класс `RouteCargoCommand` и устанавливает состояние соответствующего агрегата.

Листинг 4.14 ❖ Обработчик команды RouteCargoCommand

```
/**
 * Обработчик команды RouteCargoCommand. Устанавливает состояние агрегата Cargo и
 * регистрирует событие назначения маршрута доставки груза.
 * @param routeCargoCommand
 */
public void assignToRoute( RouteCargoCommand routeCargoCommand ) {
  this.itinerary = routeCargoCommand.getCargoItinerary();
  // Синхронная обработка согласованности внутри агрегата Cargo.
  this.delivery = delivery.updateOnRouting(this.routeSpecification, this.itinerary);
}
```

На рис. 4.25 показана диаграмма класса для реализации этого обработчика команды.

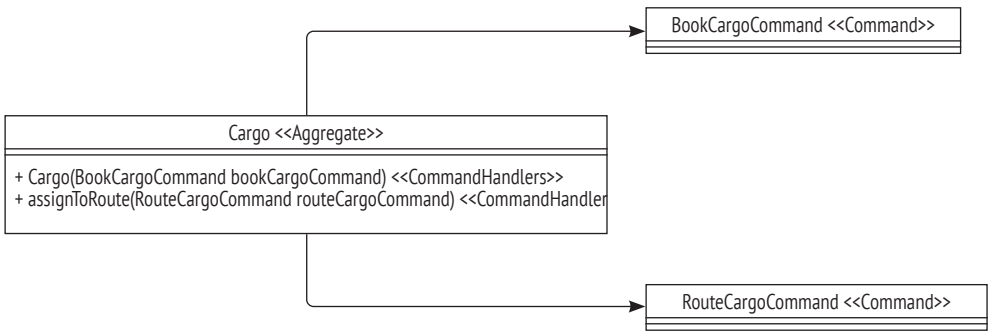

Рис. 4.25 ❖ Диаграмма класса для реализации обработчика команды

Подводя итог, отметим, что обработчики команд играют весьма важную роль в управлении состоянием агрегата в ограниченном контексте. В действительности вызовы обработчиков команд выполняются через сервисы приложения, которые будут рассматриваться в одном из следующих подразделов.

Запросы

Запросы (queries) в ограниченном контексте отвечают за представление состояния агрегата этого ограниченного контекста внешним потребителям (пользователям).

Для реализации запросов в рассматриваемом примере используются именованные запросы JPA (JPA Named Queries), т. е. запросы, которые могут быть определены в агрегате для извлечения его состояния в разнообразных формах. В листинге 4.15 приведен фрагмент кода из класса агрегата Cargo, в котором определяются необходимые запросы, доступные внешним потребителям. В рассматриваемом примере это три запроса: найти все грузы (Find All Cargos), найти груз по его идентификатору заказа (Find Cargo by its Booking Identifier) и вывести список идентификаторов заказа для всех грузов (Booking Identifiers for all Cargos).

Листинг 4.15 ❖ Именованные запросы в корневом агрегате Cargo

```
@NamedQueries( {
  @NamedQuery( name = "Cargo.findAll", query = "Select c from Cargo c" ),
  @NamedQuery( name = "Cargo.findByBookingId",
               query = "Select c from Cargo c where c.bookingId = :bookingId" ),
  @NamedQuery( name = "Cargo.findAllBookingIds",
               query = "Select c.bookingId from Cargo c" ) } )
public class Cargo{}
```

Запросы играют важную роль в представлении состояния агрегата в ограниченном контексте. В действительности вызовы и выполнение запросов происходит через сервисы приложения и классы репозиториев, которые будут рассматриваться в одном из следующих подразделов.

На этом заканчивается описание реализации запросов в модели предметной области (домена). Теперь перейдем к рассмотрению реализации событий.

События

Событие (event) в ограниченном контексте – это любая операция, которая публикует изменения состояния агрегата ограниченного контекста как событие. Поскольку команды изменяют состояние агрегата, вполне обоснованно можно предположить, что действие любой команды в ограниченном контексте приведет к возникновению соответствующего события. Подписчиками этих событий могут быть другие ограниченные контексты внутри той же предметной области (домена) или ограниченные контексты, принадлежащие каким-либо другим внешним предметным областям (доменам).

События предметной области (домена) играют центральную роль в архитектуре микросервисов, поэтому весьма важно реализовать их надежно и корректно. Распределенная природа архитектуры микросервисов диктует использование событий через механизм хореографии (choreography mechanism) для сопровождения состояния и сохранения логической целостности и согласованности транзакций между различными ограниченными контекстами приложения на основе микросервисов.

На рис. 4.26 показаны примеры событий, передаваемых между различными ограниченными контекстами в приложении Cargo Tracker.

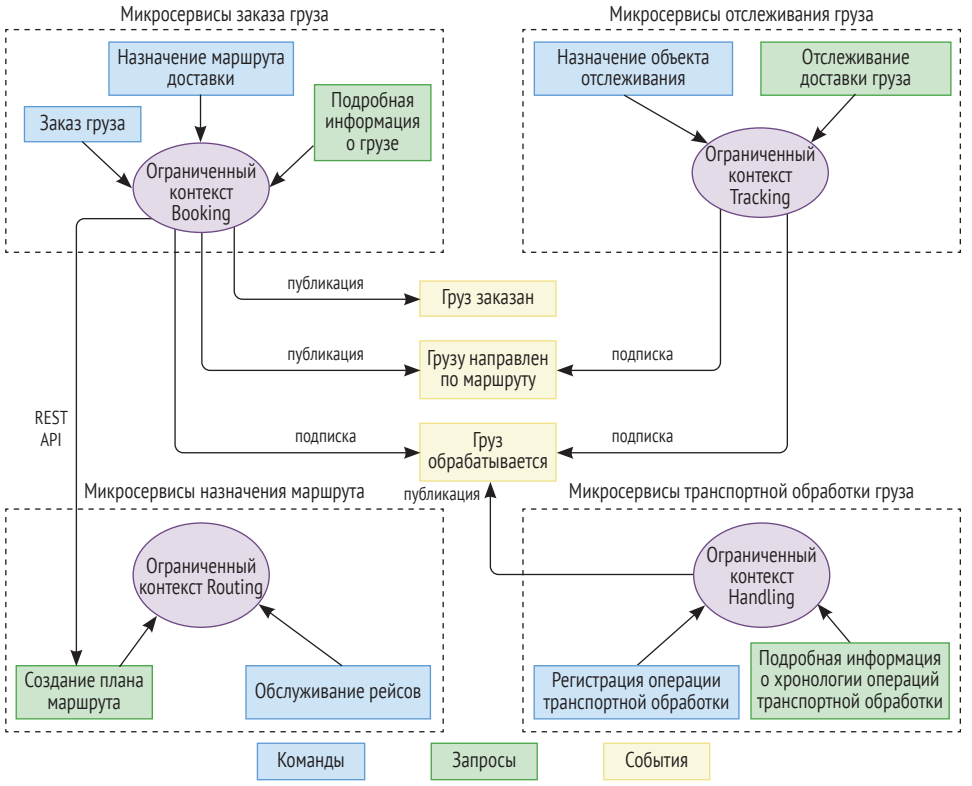

Рис. 4.26 ❖ Поток событий в архитектуре микросервисов

Рассмотрим потоки событий более подробно на конкретном примере решения бизнес-задачи.

Когда для груза назначается маршрут доставки, это означает, что теперь груз можно отслеживать, для чего в свою очередь требуется присваивание идентификатора отслеживания (Tracking Identifier) этому грузу. Назначение маршрута доставки груза выполняется в ограниченном контексте Booking (заказ груза), в то время как присваивание идентификатора отслеживания происходит в ограниченном контексте Tracking (отслеживание доставки груза). При монолитном подходе к решению этой задачи процесс назначения маршрута доставки груза и присваивания идентификатора отслеживания этого груза происходит совместно, так как можно использовать один и тот же контекст транзакций для нескольких ограниченных контекстов благодаря совместно используемой объединенной модели для процессов, сред времени выполнения и хранилищ данных.

Но в архитектуре микросервисов невозможно достичь результата тем же способом, так как в этой архитектуре нет совместно используемых элементов. При назначении маршрута доставки груза только ограниченный контекст

Booking отвечает за достоверность отображения нового назначенного маршрута в состоянии агрегата Cargo. Ограниченный контекст Tracking должен узнать об этом изменении состояния, чтобы получить возможность присваивания идентификатора отслеживания для полноценного завершения этой конкретной бизнес-задачи. Именно здесь проявляется важная роль событий предметной области (домена) и механизма хореографии событий. Если ограниченный контекст Booking может сгенерировать событие, соответствующее факту назначения маршрута агрегату Cargo, то ограниченный контекст Tracking может подписаться на это конкретное событие и при его возникновении присвоить идентификатор отслеживания для полноценного завершения этой конкретной бизнес-задачи. Механизм генерации событий и доставки событий в различные ограниченные контексты для завершения конкретных бизнес-задач представляет собой шаблон хореографии событий (event choreography pattern). Если говорить кратко, события предметной области (домена) и механизм хореографии событий помогают создавать приложения на основе микросервисов, управляемые событиями.

Можно выделить следующие четыре этапа реализации надежно функционирующей архитектуры хореографии, управляемой событиями:

- регистрацию событий предметной области (домена), которые необходимо генерировать в ограниченном контексте;
- генерацию событий предметной области (домена), которые необходимо публиковать из ограниченного контекста;
- публикацию событий, сгенерированных в ограниченном контексте;
- подписку на события, которые были опубликованы другими ограниченными контекстами.

С учетом функциональных возможностей, предоставляемых платформой MicroProfile, процесс реализации также разделяется на несколько этапов:

- генерация событий реализуется с помощью сервисов приложения;
- публикация событий реализуется с помощью исходящих сервисов;
- подписка на события обрабатывается с помощью интерфейса и входящих сервисов.

Как часть рассматриваемой здесь реализации событий предметной области (домена) будет использоваться механизм CDI Events в качестве логической инфраструктуры для публикации и подписки на события. Брокер сообщений RabbitMQ предоставляет физическую инфраструктуру для реализации механизма хореографии событий.

Поскольку в настоящий момент мы находимся на стадии реализации модели предметной области (домена), единственным из перечисленных выше этапов, рассматриваемых в этом подразделе, является создание классов событий, принимающих участие в хореографии между несколькими ограниченными контекстами. В следующих подразделах будут отдельно рассматриваться все прочие аспекты (сервисы приложения полностью охватывают реализацию генерации этих событий, исходящие сервисы отвечают за реализацию публикации событий, входящие сервисы реализуют подписку на эти события).

Классы событий в модели предметной области (домена) создаются как специализированные аннотации с использованием ключевого слова аннотации `@interface`. Использование определяемых здесь событий мы увидим в следую-

щих подразделах при реализации других областей и элементов архитектуры хореографии событий.

В листинге 4.16 показано событие `CargoBookedEvent`, реализованное как специализированная аннотация.

Листинг 4.16 ❖ Шаблонная аннотация CargoBookedEvent

```
import javax.inject.Qualifier;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

/**
 * Класс события заказа груза. Реализован как специализированная аннотация.
 */
@Qualifier
@Retention( RUNTIME )
@Target( {FIELD, PARAMETER} )
public @interface CargoBookedEvent {
}
```

В листинге 4.17 показана реализация класса события назначения маршрута доставки груза `CargoRoutedEvent`.

Листинг 4.17 ❖ Шаблонная аннотация CargoRoutedEvent

```
import javax.inject.Qualifier;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

/**
 * Класс события назначения маршрута доставки груза. В агрегате Cargo.
 */
@Qualifier
@Retention( RUNTIME )
@Target( {FIELD, PARAMETER} )
public @interface CargoRoutedEvent {
}
```

На этом завершается рассмотрение принципов реализации модели основной предметной области (домена). Далее будет подробно рассматриваться реализация сервисов модели предметной области (домена).

Сервисы модели предметной области (домена)

Сервисы модели предметной области (домена) используются по двум основным причинам. Во-первых, это обеспечение доступности состояния ограниченного контекста для внешних потребителей через корректно определенные

интерфейсы. Во-вторых, это обеспечение взаимодействия с внешними потребителями для сохранения состояния ограниченного контекста в хранилищах данных (базах данных), для публикации событий, соответствующих изменениям состояния ограниченного контекста во внешних брокерах сообщений, или для обмена данными с другими ограниченными контекстами.

Существует три типа сервисов модели предметной области (домена) для любого ограниченного контекста:

- входящие сервисы (Inbound Services), в которых реализуются корректно определенные интерфейсы, позволяющие внешним потребителям взаимодействовать с моделью предметной области (домена);
- исходящие сервисы (Outbound Services), в которых реализуются все взаимодействия с внешними репозиториями и другими ограниченными контекстами;
- сервисы приложения (Application Services), действующие как внешний уровень, связывающий модель предметной области (домена) с входящими и исходящими сервисами.

На рис. 4.27 показана реализация сервисов модели предметной области (домена).

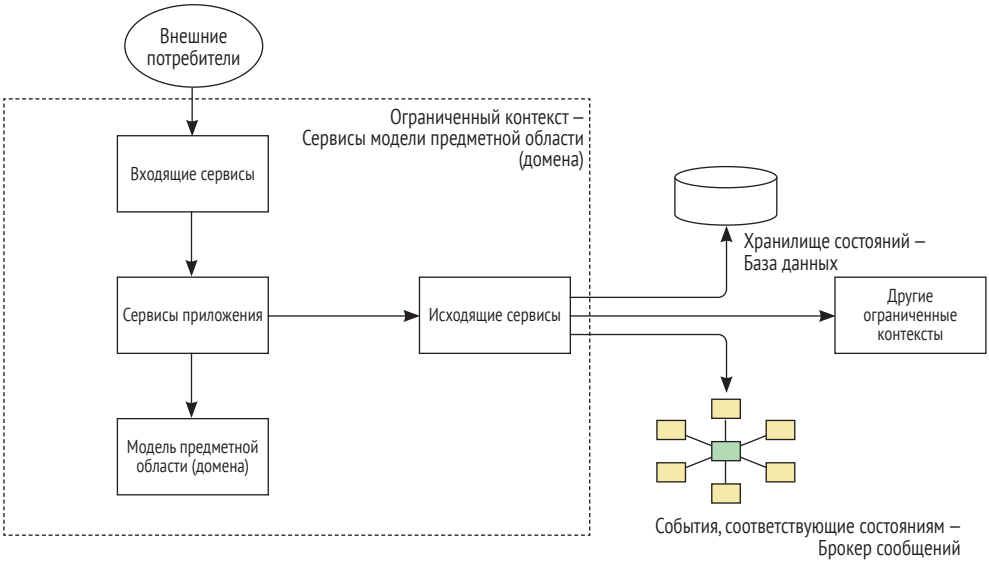

Рис. 4.27 ❖ Общая схема реализации сервисов модели предметной области (домена)

Входящие сервисы

Входящие сервисы (Inbound Services) (или входящие адаптеры (Inbound Adaptors) в соответствии с терминологией шаблона гексагональной архитектуры) действуют как внешний шлюз для модели основной предметной области (домена). Как уже было отмечено выше, здесь подразумевается реализация корректно определенных интерфейсов, позволяющих внешним потребителям взаимодействовать с моделью основной предметной области (домена).

Тип входящих сервисов зависит от типа операций, которые необходимо выполнить, чтобы разрешить взаимодействие внешних потребителей с моделью предметной области (домена).

С учетом реализации архитектурного шаблона микросервисов для приложения Cargo Tracker в рассматриваемом здесь примере определяются следующие два типа входящих сервисов:

- уровень API на основе REST, используемый внешними потребителями для вызова операций в ограниченном контексте (команды и запросы);
- уровень обработки событий на основе механизма CDI Event, который извлекает события из брокера сообщений и обрабатывает их.

REST API

Обязанности уровня REST API как доверенного представителя ограниченного контекста состоят в приеме HTTP-запросов от внешних потребителей. HTTP-запрос может быть командой или запросом. Уровень REST API обязан преобразовать внешний HTTP-запрос в модель команды или запроса, распознаваемую моделью предметной области (домена) ограниченного контекста, и передать ее на уровень сервисов приложения для дальнейшей обработки.

Еще раз обратимся к рис. 4.5, на котором подробно показаны все операции для различных ограниченных контекстов (например, заказ груза, назначение маршрута доставки груза, транспортная обработка груза, отслеживание груза и т. п.). Все эти операции будут иметь соответствующие интерфейсы REST API, которые принимают внешние запросы и обрабатывают их.

Реализация уровня REST API на платформе Eclipse MicroProfile выполняется с использованием функциональных возможностей REST, предоставляемых проектом Helidon MP на основе JAX-RS (Java API for RESTful Web Services). Из названия можно понять, что эта спецификация обеспечивает возможности для создания RESTful веб-сервисов. Проект Helidon MP предоставляет реализацию этой спецификации, таким образом, все функциональные возможности автоматически добавляются при создании каркаса проекта.

Рассмотрим подробнее пример создания REST API с использованием спецификации JAX-RS. В листинге 4.18 показан класс `CargoBookingController`, предоставляющий интерфейс REST API для команды заказа груза (Cargo Booking Command). Ниже перечислены некоторые характеристики этого класса:

- интерфейс REST API доступен по URL */cargobooking*;
- класс содержит единственный метод POST, принимающий параметр `BookCargoResource`, который содержит входящую полезную нагрузку для этого API;
- существует зависимость от сервиса приложения `CargoBookingCommandService`, который действует как внешний сервис (см. его реализацию ниже). Эта зависимость инъецирована в класс API с помощью аннотации `@Inject`, доступной как часть механизма CDI;
- класс выполняет преобразование данных ресурса `BookCargoResource` в модель команды `BookCargoModel`, используя класс утилиты ассемблера `BookCargoCommandDTOAssembler`;
- после преобразования класс делегирует процесс в сервис `CargoBookingCommandService` для дальнейшей обработки;

- класс возвращает ответ внешнему потребителю с идентификатором заказа нового груза и с сообщением состояния – успешного выполнения запроса «200 OK».

Листинг 4.18 ❖ Реализация класса CargoBookingController

```
package com.practicalddd.cargotracker.bookingms.interfaces.rest;
import com.practicalddd.cargotracker.bookingms.application.internal.\
commandservices.CargoBookingCommandService;
import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.BookingId;
import com.practicalddd.cargotracker.bookingms.interfaces.rest.dto.BookCargoResource;
import   com.practicalddd.cargotracker.bookingms.interfaces.rest.transform.\
BookCargoCommandDTOAssembler;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path( "/cargobooking" )
@ApplicationScoped
public class CargoBookingController {
  private CargoBookingCommandService cargoBookingCommandService;
  // Зависимость от сервиса приложения

  /**
   * Инъекция зависимостей (CDI)
   * @param cargoBookingCommandService
   */
  @Inject
  public CargoBookingController( CargoBookingCommandService cargoBookingCommandService ) {
    this.cargoBookingCommandService = cargoBookingCommandService;
  }
  /**
   * Метод POST для заказа груза
   * @param bookCargoResource
   */
  @POST
  @Produces( MediaType.APPLICATION_JSON )
  public Response bookCargo( BookCargoResource bookCargoResource ) {
    BookingId bookingId  = cargoBookingCommandService.bookCargo(
                  BookCargoCommandDTOAssembler.toCommandFromDTO( bookCargoResource ) );
    final Response returnValue = Response.ok().entity(bookingId).build();
    return returnValue;
  }
}
```

В листинге 4.19 показана реализация класса `BookCargoResource`.

Листинг 4.19 ❖ Реализация класса BookCargoResource для класса контроллера

```
package com.practicalddd.cargotracker.bookingms.interfaces.rest.dto;

import java.time.LocalDate;
```

```
/**
 * Resource class for the Book Cargo Command API
 */
public class BookCargoResource {
  private int bookingAmount;
  private String originLocation;
  private String destLocation;
  private LocalDate destArrivalDeadline;

  public BookCargoResource() {}
  public BookCargoResource( int bookingAmount, String originLocation, String destLocation,
                            LocalDate destArrivalDeadline ) {
    this.bookingAmount = bookingAmount;
    this.originLocation = originLocation;
    this.destLocation = destLocation;
    this.destArrivalDeadline = destArrivalDeadline;
  }

  public void setBookingAmount( int bookingAmount ) {
    this.bookingAmount = bookingAmount;
  }

  public int getBookingAmount() {
    return this.bookingAmount;
  }

  public String getOriginLocation() { return originLocation; }
  public void setOriginLocation( String originLocation ) {
    this.originLocation = originLocation;
  }

  public String getDestLocation() { return destLocation; }
  public void setDestLocation( String destLocation ) { this.destLocation = estLocation; }
  public LocalDate getDestArrivalDeadline() { return destArrivalDeadline; }
  public void setDestArrivalDeadline( LocalDate destArrivalDeadline ) {
    this.destArrivalDeadline = destArrivalDeadline;
  }
}
```

В листинге 4.20 показана реализация класса `BookCargoCommandDTOAssembler`.

Листинг 4.20 ❖ Реализация класса BookCargoCommandDTOAssembler

```
package com.practicalddd.cargotracker.bookingms.interfaces.rest.transform;

import com.practicalddd.cargotracker.bookingms.domain.model.commands.BookCargoCommand;
import com.practicalddd.cargotracker.bookingms.interfaces.rest.dto.BookCargoResource;

/**
 * Класс ассемблера (сборки) для преобразования данных BookCargoResource
 * в модель BookCargo.
 */
public class BookCargoCommandDTOAssembler {
  /**
   * Статический метод в классе Assembler.
   * @param bookCargoResource
   * @return BookCargoCommand Model
```

```
   */
  public static BookCargoCommand toCommandFromDTO( BookCargoResource bookCargoResource ) {
    return new BookCargoCommand(
                bookCargoResource.getBookingAmount(),
                bookCargoResource.getOriginLocation(),
                bookCargoResource.getDestLocation(),
                java.sql.Date.valueOf(bookCargoResource.getDestArrivalDeadline() ) );
  }
}
```

В листинге 4.21 показана реализация класса `BookCargoCommand`.

Листинг 4.21 ❖ Реализация класса BookCargoCommand

```
package com.practicalddd.cargotracker.bookingms.domain.model.commands;

import java.util.Date;

/**
 * Класс команды BookCargoCommand.
 */
public class BookCargoCommand {
  private int bookingAmount;
  private String originLocation;
  private String destLocation;
  private Date destArrivalDeadline;

  public BookCargoCommand() {}
  public BookCargoCommand( int bookingAmount, String originLocation, String destLocation,
                           Date destArrivalDeadline ) {
    this.bookingAmount = bookingAmount;
    this.originLocation = originLocation;
    this.destLocation = destLocation;
    this.destArrivalDeadline = destArrivalDeadline;
  }

  public void setBookingAmount( int bookingAmount ) {
    this.bookingAmount = bookingAmount;
  }

  public int getBookingAmount() {
    return this.bookingAmount;
  }

  public String getOriginLocation() { return originLocation; }

  public void setOriginLocation( String originLocation ) {
    this.originLocation = originLocation;
  }

  public String getDestLocation() { return destLocation; }

  public void setDestLocation( String destLocation ) {
    this.destLocation = destLocation;
  }

  public Date getDestArrivalDeadline() { return destArrivalDeadline; }

  public void setDestArrivalDeadline( Date destArrivalDeadline ) {
```

```
    this.destArrivalDeadline = destArrivalDeadline;
  }
}
```

На рис. 4.28 показана диаграмма класса сервиса обработки команды заказа груза для рассматриваемого здесь примера реализации.

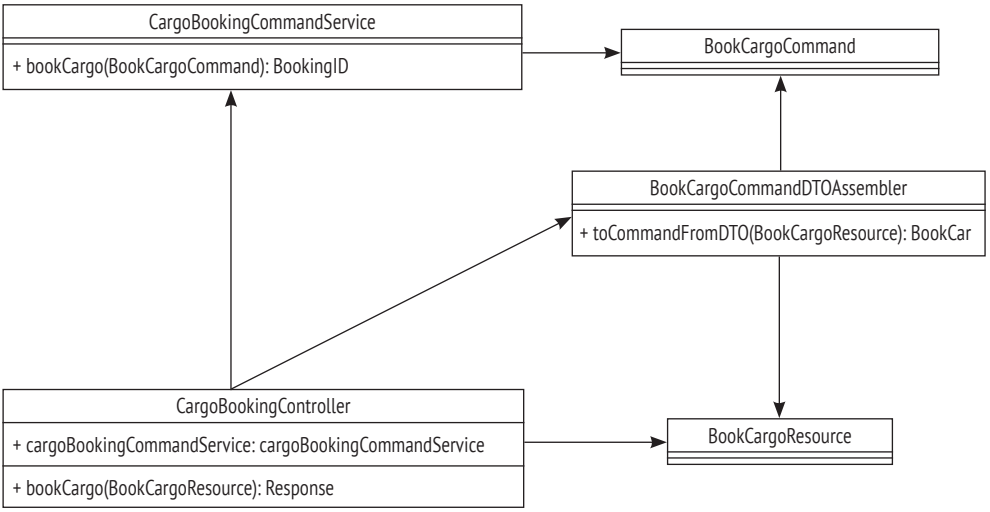

Рис. 4.28 ❖ Диаграмма класса для реализации интерфейса REST API

Все рассматриваемые здесь реализации входящих интерфейсов REST API используют один и тот же подход, который показан на рис. 4.29.

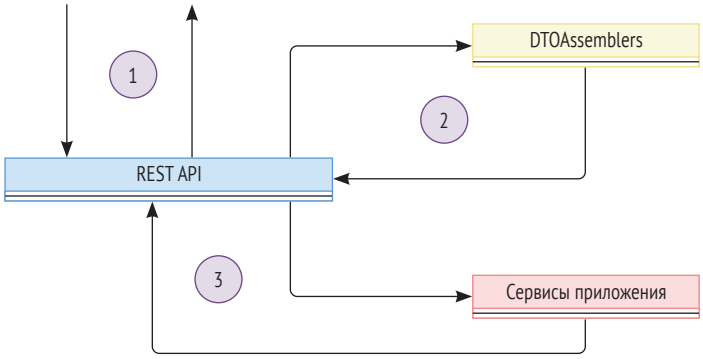

Рис. 4.29 ❖ Общая схема процесса реализации входящих сервисов

1. Входящий внешний запрос на выполнение команды или (внутреннего) запроса поступает в интерфейс REST API. Классы API реализуются с использованием функциональных возможностей JAX-RS, предоставляемых проектом Helidon MP.
2. Класс REST API использует утилиту Assembler как компонент для преобразования формата данных ресурса в формат данных команды или запроса, требуемый моделью предметной области (домена).
3. Данные команды или запроса передаются в сервисы приложения для дальнейшей обработки.

Обработчики событий

Другим типом интерфейсов, существующих в ограниченных контекстах, являются обработчики событий (Event Handlers). В любом ограниченном контексте они отвечают за обработку событий, которые в той или иной степени важны для данного конкретного ограниченного контекста. События генерируются другими ограниченными контекстами в том же приложении. Объекты типа `EventHandler` создаются в ограниченном контексте, который является подписчиком и размещается на уровне *inbound/interface*. Обработчики событий принимают событие (Event) вместе с данными полезной нагрузки этого события и обрабатывают их в форме обычной операции.

В листинге 4.22 показан обработчик событий `CargoRoutedEventHandler`, который расположен в ограниченном контексте Tracking (отслеживание груза). Он наблюдает за событием `CargoRoutedEvent` и принимает данные `CargoRoutedEventData` как полезную нагрузку.

Листинг 4.22 ❖ Реализация обработчика событий CargoRouted

```
package com.practicalddd.cargotracker.trackingms.interfaces.events;

import com.practicalddd.cargotracker.shareddomain.events.CargoRoutedEvent;
import com.practicalddd.cargotracker.shareddomain.events.CargoRoutedEventData;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;

@ApplicationScoped
public class CargoRoutedEventHandler {
  public void receiveEvent( @Observes @CargoRoutedEvent CargoRoutedEventData eventData ) {
        // Обработка этого события.
  }
}
```

При реализации всех обработчиков событий в рассматриваемом здесь примере используется один и тот же подход, показанный на рис. 4.30.

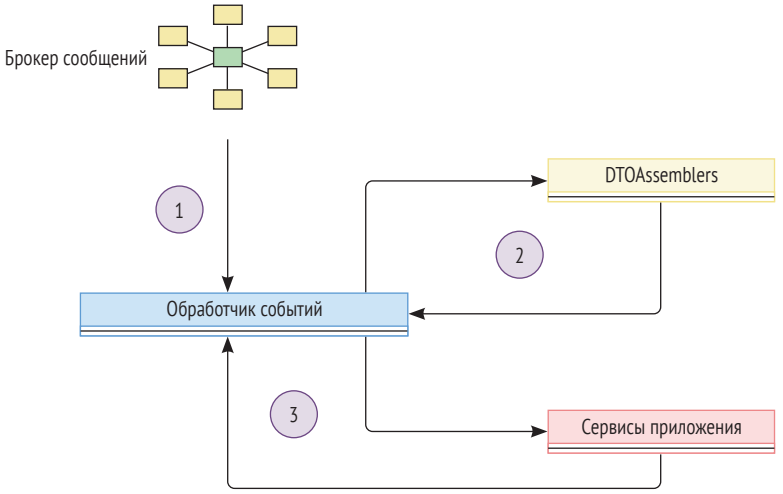

Рис. 4.30 ❖ Общая схема реализации обработчиков событий

1. Обработчики событий принимают входящие события от брокера сообщений.
2. Обработчики событий используют утилиту Assembler как компонент для преобразования формата данных ресурса в формат данных команды, требуемый моделью предметной области (домена).
3. Данные команды передаются в сервисы приложения для дальнейшей обработки.

Реализация процесса отображения входящих событий в соответствующие физические очереди брокера сообщений (RabbitMQ) рассматривается как часть реализации уровня исходящих сервисов – брокера сообщений.

Сервисы приложения

Сервисы приложения действуют как внешний связующий уровень или порт между входящими/исходящими сервисами и моделью основной предметной области (домена) в ограниченном контексте.

В ограниченном контексте сервисы приложения отвечают за прием запросов от входящих сервисов и делегирование их соответствующим сервисам, т. е. команды передаются в сервисы команд, запросы – сервисам запросов, а внешние запросы (requests) для обмена данными с другими ограниченными контекстами – в исходящие сервисы. Как часть процесса взаимодействия с командами, запросами и внешними ограниченными контекстами, сервисы приложения могут быть затребованы для обмена данными с репозиториями, брокерами сообщений или другими ограниченными контекстами. Исходящие сервисы используются для помощи при таком обмене данными.

Поскольку спецификация MicroProfile не предоставляет возможность генерации событий предметной области (домена) непосредственно из модели предметной области (домена), в рассматриваемом здесь примере сервисы приложения используются для генерации событий предметной области (до-

мена). Эти события публикуются в брокерах сообщений с использованием исходящих сервисов.

Классы сервисов приложения реализуются с использованием механизма CDI Managed Beans. Проект Helidon MP предоставляет реализацию механизма CDI, и эта функциональная возможность автоматически добавляется при создании каркаса проекта.

На рис. 4.31 показаны обязанности сервисов приложения.

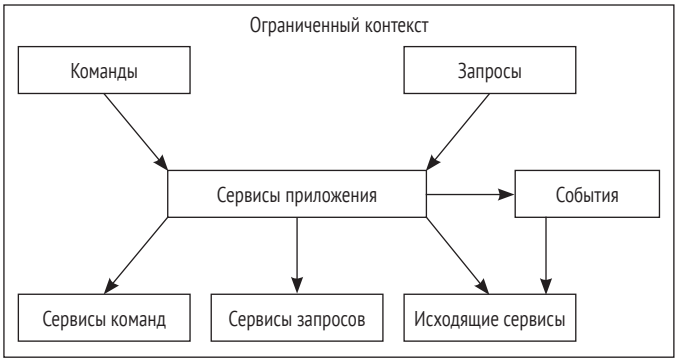

Рис. 4.31 ❖ Обязанности сервисов приложения

Сервисы приложения: делегирование команд и запросов

Как часть рассматриваемой здесь области ответственности, сервисы приложений в ограниченном контексте принимают внешние запросы (requests) на обработку команд и запросов. Обычно такие внешние запросы приходят из входящих сервисов (уровень API). Как часть этого процесса обработки, сервисы приложения сначала используют обработчики команд и запросов (см. соответствующий раздел по модели предметной области (домена)) модели предметной области (домена) для установки состояния или запроса о состоянии. Затем сервисы приложения используют исходящие сервисы для сохранения состояния или для выполнения запросов о состоянии агрегата.

Рассмотрим пример класса сервисов приложения, предназначенный для делегирования команд (Command Delegator), а именно класс сервисов приложения для команды заказа груза (Cargo Booking Command), который обрабатывает все команды, относящиеся к ограниченному контексту Booking. Кроме того, будет подробно рассмотрена одна из команд – команда заказа груза (Book Cargo), представляющая собой инструкцию для заказа нового груза. Характеристики класса и команды описаны ниже:

- класс сервисов приложения реализуется как компонент CDI Bean с назначением ему конкретной области видимости (в данном примере областью видимости является все приложение `@Application`);
- класс сервисов приложения обеспечивается всем необходимыми зависимостями с помощью аннотации `@Inject`. В рассматриваемом примере класс `CargoBookingCommandApplicationService` зависит от класса репозитория исходящего сервиса (`CargoRepository`), который используется для обеспе-

чения персистентности нового создаваемого груза. Также существует зависимость от события CargoBookedEvent, которое должно быть сгенерировано сразу после обеспечения персистентности созданного груза. CargoBookedEvent – это шаблонный класс, который содержит объект груза Cargo как полезную нагрузку события. В одном из следующих подразделов мы более подробно опишем события предметной области (домена), но в настоящий момент достаточно этого замечания;

- сервисы приложения делегируют обработку в метод bookCargo. Перед началом обработки этим методом сервисы приложения убеждаются в том, что открыта новая транзакция посредством аннотации @Transactional;
- класс сервисов приложения сохраняет новый заказанный груз в базе данных MySQL Booking (таблица CARGO) и генерирует событие CargoBookedEvent;
- класс сервисов приложения возвращает ответ во входящий сервис с указанием идентификатором заказа (Booking Identifier) нового заказанного груза.

В листинге 4.23 показана реализация класса сервиса приложения для обработки команды заказа нового груза.

Листинг 4.23 ❖ Реализация класса CargoBookingCommandApplicationService

```
package com.practicalddd.cargotracker.bookingms.application.internal.commandservices;

import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.BookingId;
import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.Cargo;
import com.practicalddd.cargotracker.bookingms.domain.model.commands.BookCargoCommand;
import com.practicalddd.cargotracker.bookingms.domain.model.entities.Location;
import com.practicalddd.cargotracker.bookingms.domain.model.events.CargoBookedEvent;
import com.practicalddd.cargotracker.bookingms.domain.model.valueobjects.BookingAmount;
import com.practicalddd.cargotracker.bookingms.domain.model.valueobjects.
RouteSpecification;
import com.practicalddd.cargotracker.bookingms.infrastructure.repositories.jpa.
CargoRepository;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Event;
import javax.inject.Inject;
import javax.transaction.Transactional;

/**
 * Класс сервиса приложения для команды заказа груза.
 */
@ApplicationScoped // Область видимости компонента CDI Managed Bean. Область видимости
                   // Application определяет единственный экземпляр для этого класса
                   // сервиса.
public class CargoBookingCommandService  // Компонент CDI Managed Bean
{
  @Inject
  private CargoRepository cargoRepository; // Исходящий сервис для соединения с экземпляром
                                           // базы данных MySQL Booking соответствующего
                                           // ограниченного контекста.
  @Inject
  @CargoBookedEvent
```

```
  private Event<Cargo> cargoBooked; // Событие, которое необходимо сгенерировать при заказе
                                    // нового груза.
  /**
   * Метод сервиса команд для заказа нового груза.
   * @return BookingId of the Cargo
   */
  @Transactional // Инициализация транзакции.
  public BookingId bookCargo( BookCargoCommand bookCargoCommand ) {
    BookingId bookingId = cargoRepository.nextBookingId();
    RouteSpecification routeSpecification = new RouteSpecification(
                new Location( bookCargoCommand.getOriginLocation() ),
                new Location( bookCargoCommand.getDestLocation() ),
                bookCargoCommand.getDestArrivalDeadline() );
    BookingAmount bookingAmount = new BookingAmount( bookCargoCommand.getBookingAmount() );
    Cargo cargo = new Cargo(
                bookingId,
                bookingAmount,
                routeSpecification );
    cargoRepository.store( cargo ); // Сохранение груза.
    cargoBooked.fire( cargo ); // генерация события заказа нового груза.
    return bookingId;
  }

  // Далее следуют реализации всех прочих команд для ограниченного контекста Booking.
}
```

В листинге 4.24 показана реализация класса сервисов приложений для запроса по заказу груза, который обслуживает все запросы, относящиеся к ограниченному контексту Booking (заказ груза). Реализация аналогична реализации класса сервисов приложений для команды заказа нового груза за исключением того, что здесь не генерируется какое-либо событие предметной области (домена), так как это всего лишь запрос состояния ограниченного контекста, который не изменяет это состояние.

Листинг 4.24 ❖ Реализация класса запроса CargoBookingQueryApplicationService

```
package com.practicalddd.cargotracker.bookingms.application.internal.queryservices;

import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.BookingId;
import   com.practicalddd.cargotracker.bookingms.domain.model.aggregates.Cargo;
import com.practicalddd.cargotracker.bookingms.infrastructure.repositories.jpa.
CargoRepository;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import javax.transaction.Transactional;
import java.util.List;

/**
 * Сервис приложения, который обрабатывает все запросы, относящиеся к ограниченному
 * контексту Booking.
 */
@ApplicationScoped
public class CargoBookingQueryService {
  @Inject
```

```
  private CargoRepository cargoRepository; // Инъекция зависимостей.
  /**
   * Поиск всех (заказанных) грузов.
   * @return List<Cargo>
   */
  @Transactional
  public List<Cargo> findAll() {
    return cargoRepository.findAll();
  }

  /**
   * Список всех идентификаторов заказанных грузов.
   * @return List<BookingId>
   */
  public List<BookingId> getAllBookingIds() {
    return cargoRepository.getAllBookingIds();
  }

  /**
   * Поиск конкретного заказанного груза по идентификатору его заказа BookingId.
   * @param bookingId
   * @return Cargo
   */
  public Cargo find( String bookingId ) {
    return cargoRepository.find( new BookingId( bookingId ) );
  }
}
```

На рис. 4.32 показана диаграмма класса для реализации, приведенной в листинге 4.24.

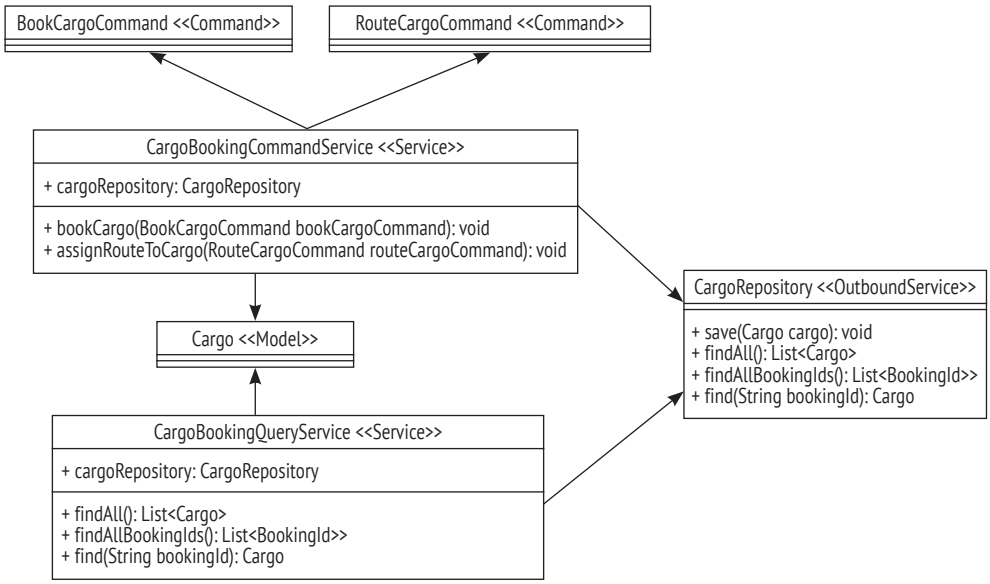

Рис. 4.32 ❖ Диаграмма класса сервисов приложения, делегирующего команды или запросы

При реализации всех сервисов приложения (команд и запросов) в рассматриваемом здесь примере используется один и тот же подход, показанный на рис. 4.33.

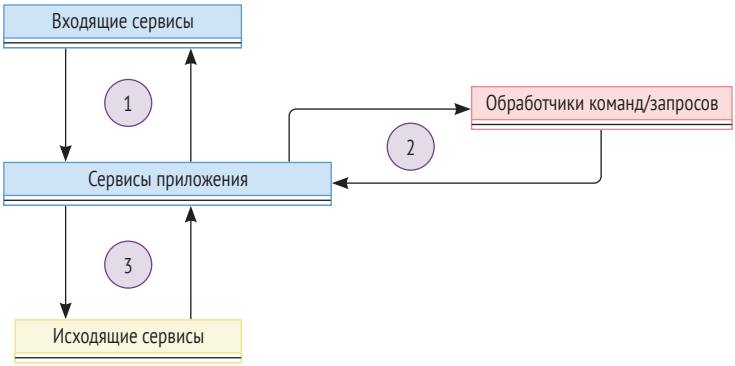

Рис. 4.33 ❖ Общая схема реализации сервисов приложения как делегирующих команды и запросы

1. Внешний запрос (request) на выполнение команды/запроса поступает в сервисы приложения ограниченного контекста. Этот внешний запрос передается из входящих сервисов. Классы сервисов приложения реализуются как компоненты CDI Managed Beans, содержащие все необходимые зависимости, инъецируемые с помощью CDI аннотации Injection. Классы сервисов приложения имеют определенную область видимости и отвечают за создание контекста транзакции для начала выполнения запрашиваемой операции.
2. Сервисы приложения полагаются на обработчики команд и запросов при установлении и запросе состояния агрегата.
3. Как часть общего процесса выполнения запрашиваемой операции, сервисы приложения требуют обеспечения взаимодействия с внешними репозиториями. Для обеспечения этих взаимодействий необходимы исходящие сервисы.

Сервисы приложения: генерация событий предметной области (домена)

Другая роль, которую играют сервисы приложения, – генерация событий предметной области (домена) при обработке ограниченным контекстом команды.

В предыдущей главе события предметной области (домена) были реализованы с использованием модели CDI 2.0 Eventing. Шина событий на основе модели оповещения/отслеживания событий работает как механизм регулирования и координации для производителей и потребителей событий. В монолитной версии использовалась исключительно внутренняя реализация шины событий (Event Bus), а события генерировались и потреблялись в одном и том же потоке выполнения.

В среде микросервисов такой метод работать не будет. Поскольку каждый микросервис развертывается (и функционирует) отдельно и независимо от других, необходим централизованный брокер сообщений, который регулирует поток событий между производителями и потребителями в различных огра-

ниченных контекстах/микросервисах, как показано на рис. 4.34. В рассматриваемой здесь реализации централизованным брокером сообщений является RabbitMQ.

При использовании платформы Eclipse MicroProfile и механизма CDI Events работа с событиями в основном становится состоящей из двух компонентов:

- генерации событий CDI Events и их публикации как сообщений в брокере RabbitMQ;
- наблюдении за событиями CDI Events и их потреблении (извлечении) как сообщений из брокера RabbitMQ.

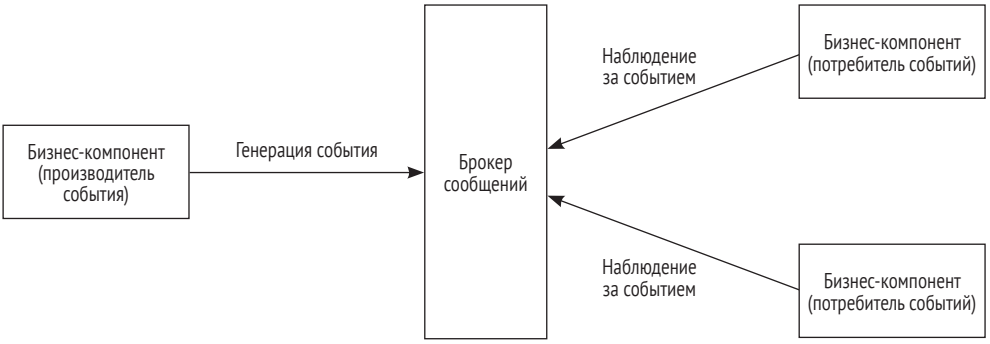

Рис. 4.34 ❖ Общая схема обработки событий предметной области (домена)

Ранее уже рассматривался бизнес-вариант для ограниченного контекста Tracking (отслеживание груза), требующий подписки на событие Cargo Routed (маршрут назначен для груза) из ограниченного контекста Booking (заказ груза), чтобы присвоить идентификатор отслеживания (Tracking Identifier) новому заказанному грузу. Рассмотрим подробнее реализацию этого бизнес-варианта, чтобы продемонстрировать процесс публикации событий сервисами приложений.

В листинге 4.25 показан фрагмент кода из класса `CargoBookingCommandService`, который обрабатывает команду назначения маршрута для нового груза, затем генерирует событие `CargoRouted`.

Листинг 4.25 ❖ Генерация события CargoRouted из класса сервиса приложения

```
package com.practicalddd.cargotracker.bookingms.application.internal.commandservices;

import javax.enterprise.event.Event; // Механизм событий CDI.

/**
 * Класс сервиса приложения для команды заказа груза.
 */
@ApplicationScoped
public class CargoBookingCommandService {
  @Inject
  private CargoRepository cargoRepository;
  @Inject
  @CargoRoutedEvent // Специализированная аннотация для события CargoRouted.
  private Event<CargoRoutedEventData> cargoRouted; // Событие, которое должно быть
                            // сгенерировано при назначении маршрута для данного груза.
```

```
  /**
   * Метод сервиса команд для назначения маршрута для нового груза.
   * @param routeCargoCommand
   */
  @Transactional
  public void assignRouteToCargo( RouteCargoCommand routeCargoCommand ) {
    Cargo cargo = cargoRepository.find( new BookingId( routeCargoCommand.
getCargoBookingId() ) );
    CargoItinerary cargoItinerary =
      externalCargoRoutingService.fetchRouteForSpecification(
        new RouteSpecification(
              new Location( routeCargoCommand.getOriginLocation() ),
              new Location( routeCargoCommand.getDestinationLocation() ),
              routeCargoCommand.getArrivalDeadline()
        ) );
      routeCargoCommand.setCargoItinerary( cargoItinerary );
      cargo.assignToRoute( routeCargoCommand );
      cargoRepository.store( cargo );
      cargoRouted.fire( new CargoRoutedEventData( routeCargoCommand.getCargoBookingId() )
);
  }
}
```

В процессе реализации механизма генерации событий CDI Events необходимо выделить следующие три этапа генерации событий и их полезной нагрузки:

- инъекцию сервисов приложения вместе с событием, которое должно быть сгенерировано. Для этого используется специализированная аннотация для конкретного события, реализованного как часть операций модели предметной области (домена) (Events). В рассматриваемом здесь примере инъецируется специализированная аннотация `CargoRoutedEvent` в сервис команд `CargoBookingCommandService`;
- создается объект данных полезной нагрузки события, который будет содержать полезную нагрузку этого конкретного публикуемого события. В рассматриваемом здесь примере это объект `CargoRoutedEventData`;
- метод `fire()`, предоставляемый механизмом CDI, используется для генерации события и упаковки полезной нагрузки, чтобы передать ее вместе с этим событием.

На рис. 4.35 показана диаграмма класса для реализации сервиса приложения, генерирующего событие предметной области (домена).

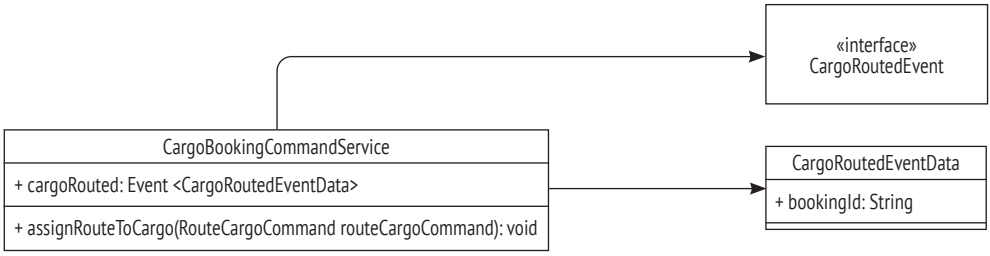

Рис. 4.35 ❖ Диаграмма класса сервиса приложения, генерирующего событие предметной области (домена)

Все реализации сервисов приложения из рассматриваемого здесь примера, отвечающие за генерацию событий предметной области (домена), следуют одной и той же методике, которая показана на рис. 4.36.

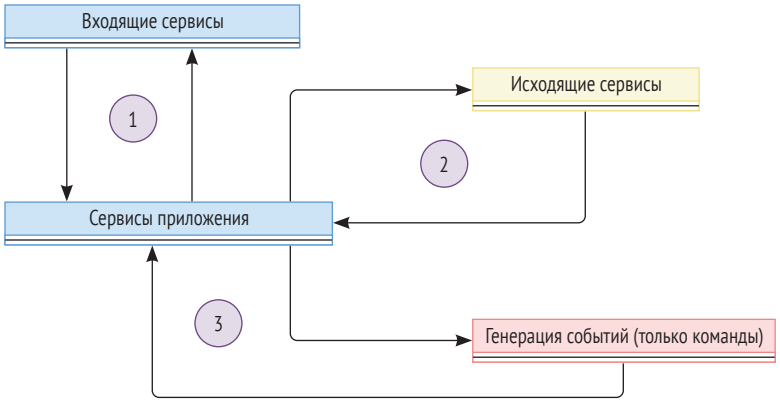

Рис. 4.36 ❖ Общая схема реализации сервисов приложения, отвечающих за генерацию событий предметной области (домена)

На этом завершается рассмотрение реализации механизма генерации событий предметной области (домена). Но реализация механизма генерации событий предметной области (домена) пока остается на логическом уровне. Теперь необходимо публиковать эти события в физически существующем брокере сообщений (в нашем случае это брокер сообщений RabbitMQ) для завершенности архитектуры механизма сообщений. Для этой цели используются связующие классы (Binder Classes), которые рассматриваются в следующем разделе (подраздел о взаимодействии с брокером сообщений).

Исходящие сервисы

Исходящие сервисы предоставляют функциональные возможности для взаимодействия с сервисами, являющимися внешними по отношению к ограниченному контексту. Таким внешним сервисом может быть хранилище данных, где содержится состояние агрегата ограниченного контекста или брокер сообщений, в котором публикуется состояние агрегата. К внешним сервисам также относится взаимодействие с другим ограниченным контекстом.

На рис. 4.37 показаны обязанности исходящих сервисов. Исходящие сервисы принимают внешние запросы (requests) на обмен данными с внешними сервисами как части операции (команды, запросы, события). Исходящие сервисы используют API (API персистентности, REST API, API брокера) в соответствии с типом внешнего сервиса, с которым необходимо взаимодействие.

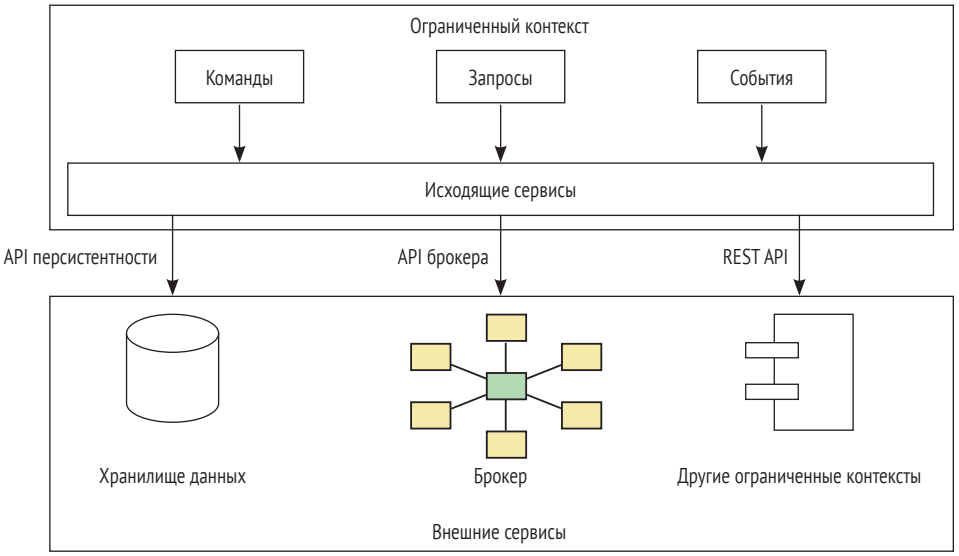

Рис. 4.37 ❖ Исходящие сервисы

Рассмотрим подробнее реализации различных типов исходящих сервисов.

Исходящие сервисы: репозитории

Исходящие сервисы для доступа к базе данных реализуются как классы `Repository`. Класс репозитория создается для конкретного агрегата и выполняет все операции базы данных для этого агрегата, в том числе следующие:

- обеспечение персистентности нового агрегата и всех связанных с ним объектов;
- обновление агрегата и всех связанных с ним объектов;
- обслуживание запросов к агрегату и к связанным с ним объектам.

Рассмотрим пример класса репозитория CargoRepository, который выполняет все операции базы данных, связанные с агрегатом Cargo:

- класс репозитория реализуется как компонент CDI Bean с назначением ему области видимости (в данном примере область видимости `@Application`);
- поскольку будет использоваться JPA как механизм взаимодействия с конкретным экземпляром базы данных, класс репозитория обеспечивается менеджером ресурса объекта, управляемым JPA. Менеджер ресурса объекта обеспечивает взаимодействие с базой данных, предоставляя для этого уровень инкапсуляции. Менеджер объекта инъецируется с использованием аннотации `@PersistenceContext`. Эта аннотация связана с файлом persistence.xml, содержащим информацию о соединении с реально существующей физической базой данных. Ранее уже рассматривалась реализация файла persistence.xml как части процесса настройки проекта Helidon MP;

- класс репозитория использует методы, предоставляемые менеджером объекта (`persist()`) для обеспечения персистентности и обновления экземпляров агрегата Cargo;
- класс репозитория использует методы, предоставляемые менеджером объекта для создания именованных запросов JPA (`createNamedQueries()`) и их выполнения, чтобы возвращать результаты.

В листинге 4.26 показана реализация класса `CargoRepository`.

Листинг 4.26 ❖ Реализация класса CargoRepository

```
package com.practicalddd.cargotracker.bookingms.infrastructure.repositories.jpa;

import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.BookingId;
import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.Cargo;

import javax.enterprise.context.ApplicationScoped;
import javax.persistence.EntityManager;
import javax.persistence.NoResultException;
import javax.persistence.PersistenceContext;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Класс репозитория для агрегата Cargo. Выполняет все операции с репозиторием, связанные
 * с состоянием агрегата Cargo.
 */
@ApplicationScoped
public class CargoRepository {
  private static final long serialVersionUID = 1L;
  private static final Logger logger = Logger.getLogger( CargoRepository.class.getName() );
  @PersistenceContext( unitName = "bookingms" )
  private EntityManager entityManager;
  /**
   * Возвращает агрегат Cargo по идентификатору его заказа Booking Identifier.
   * @param bookingId
   * @return
   */
  public Cargo find(BookingId bookingId) {
    Cargo cargo;
    try {
      cargo = entityManager.createNamedQuery( "Cargo.findByBookingId", Cargo.class )
              .setParameter( "bookingId", bookingId )
              .getSingleResult();
    } catch( NoResultException e ) {
        logger.log( Level.FINE, "Find called on non-existant Booking ID.", e );
```

```
        cargo = null;
    }
    return cargo;
  }

  /**
   * Сохранение агрегата Cargo.
   * @param cargo
   */
  public void store( Cargo cargo ) {
    entityManager.persist( cargo );
  }

  /**
   * Получение следующего идентификатора заказа груза.
   * @return
   */
  public BookingId nextBookingId() {
    String random = UUID.randomUUID().toString().toUpperCase();
    return new BookingId( random.substring( 0, random.indexOf("-") ) );
  }

  /**
   * Find all Cargo Aggregates
   * @return
   */
  public List<Cargo> findAll() {
    return entityManager.createNamedQuery( "Cargo.findAll",Cargo.class )
                        .getResultList();
  }

  /**
   * Получение всех идентификаторов заказа груза
   * @return
   */
  public List<BookingId> getAllBookingIds() {
    List<BookingId> bookingIds = new ArrayList<BookingId>();
    try {
      bookingIds = entityManager.createNamedQuery(
                   "Cargo.getAllTrackingIds", BookingId.class ).getResultList();
    } catch( NoResultException e ) {
        logger.log( Level.FINE, "Unable to get all tracking IDs", e );
    }
    return bookingIds;
  }
}
```

Все реализации классов репозитория в рассматриваемом здесь примере следуют одной и той же методике, показанной на рис. 4.38.

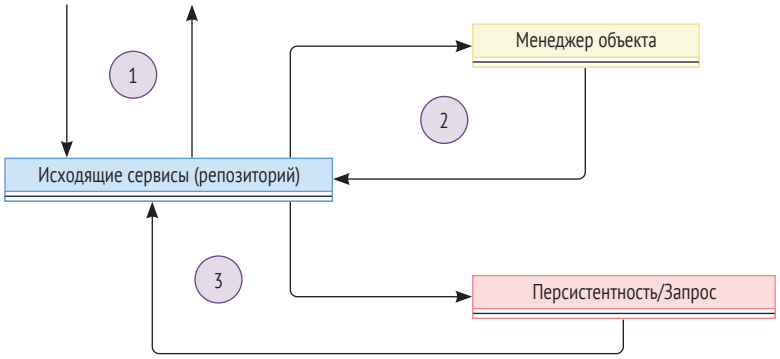

Рис. 4.38 ❖ Общая схема реализации класса репозитория

1. Репозитории принимают внешние запросы (requests) на изменение/запрос состояния агрегата.
2. Репозитории используют менеджер объекта для выполнения операций базы данных, связанных с конкретным агрегатом (сохранение, запрос состояния).
3. Менеджер объекта выполняет требуемую операцию и возвращает результаты в класс репозитория.

Исходящие сервисы: REST API

Использование REST API как режима обмена данными между микросервисами – это достаточно часто встречающееся техническое требование. Ранее мы рассматривали хореографию событий как механизм для достижения этой цели, но иногда синхронного вызова между ограниченными контекстами вполне достаточно для решения той же задачи.

Рассмотрим этот подход на конкретном примере. Как часть процесса заказа нового груза необходимо сформировать для этого груза план маршрута доставки, зависящий от спецификации назначенного маршрута. Данные, требуемые для генерации оптимального плана маршрута доставки, обрабатываются как часть ограниченного контекста Routing (назначение маршрута), в котором выполняется обработка перемещений, планов и графиков для транспортных средств (судов). При этом требуется, чтобы сервис заказа груза ограниченного контекста Booking выполнил исходящий вызов сервиса маршрутизации ограниченного контекста Routing, который предоставляет REST API для извлечения всех возможных планов маршрута доставки, зависимых от спецификации маршрута этого груза.

Этот процесс показан на рис. 4.39.

Но при этом возникает затруднение с точки зрения модели предметной области (домена). Агрегат Cargo ограниченного контекста Booking содержит представление плана маршрута доставки как объект `CargoItinerary`, в то время как в ограниченном контексте Routing план маршрута доставки представлен как объект `TransitPath`. Таким образом, вызов между этими двумя ограниченными контекстами потребует преобразования типов объектов между соответствующими моделями предметных областей (доменов).

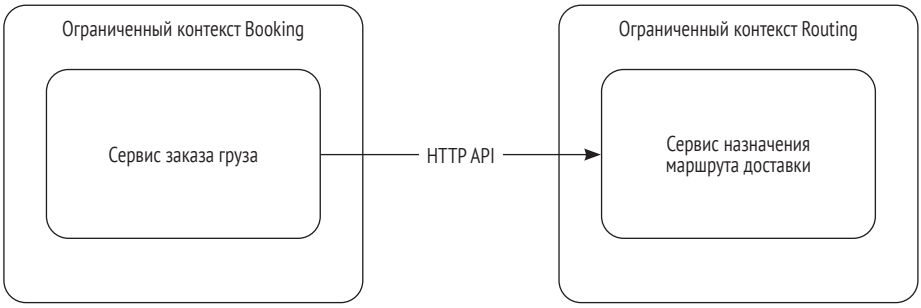

Рис. 4.39 ❖ HTTP-вызов между двумя ограниченными контекстами

Такое преобразование обычно выполняется на уровне Anti-corruption (защита от повреждений), который действует как упрощенный шлюз (bridge) для обмена данными между двумя ограниченными контекстами, как показано на рис. 4.40.

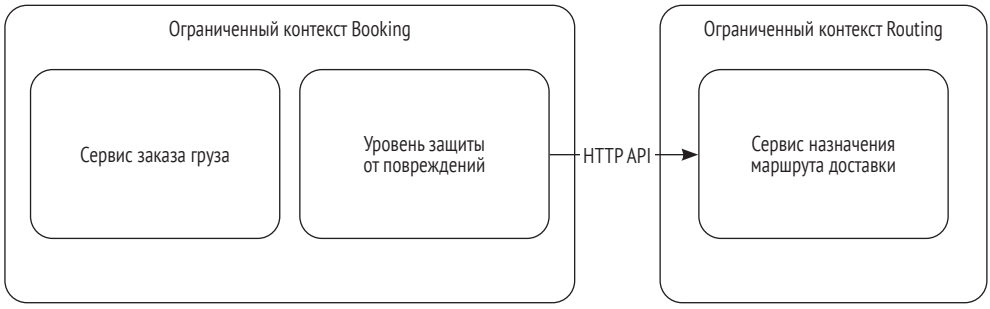

Рис. 4.40 ❖ Уровень защиты от повреждений между двумя ограниченными контекстами

Ограниченный контекст Booking полагается на функциональные возможности типобезопасной работы с клиентом MicroProfile Type Safe Rest Client, предоставляемые проектом Helidon MP для вызова REST API сервиса назначения маршрута доставки.

Чтобы лучше понять изложенную концепцию, рассмотрим полную ее реализацию.

Первый шаг – реализация REST API сервиса назначения маршрута доставки груза. Для этого используются стандартные функциональные возможности JAX-RS, которые уже были реализованы в предыдущей главе. В листинге 4.27 показана реализация REST API сервиса назначения маршрута доставки груза:

Листинг 4.27 ❖ Реализация контроллера CargoRouting

```
package com.practicalddd.cargotracker.routingms.interfaces.rest;
import com.practicalddd.cargotracker.TransitPath;
import com.practicalddd.cargotracker.routingms.application.internal.CargoRoutingService;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
```

```
import javax.ws.rs.*;
@Path( "/cargoRouting" )
@ApplicationScoped
public class CargoRoutingController {
  private CargoRoutingService cargoRoutingService; // Зависимость от сервиса приложения.
  /**
   * Обеспечение зависимостей.
   * @param cargoRoutingService
   */
  @Inject
  public CargoRoutingController( CargoRoutingService cargoRoutingService ) {
    this.cargoRoutingService = cargoRoutingService;
  }
  /**
   *
   * @param originUnLocode
   * @param destinationUnLocode
   * @param deadline
   * @return TransitPath - Оптимальный план маршрута для заданной спецификации маршрута.
   */
  @GET
  @Path( "/optimalRoute" )
  @Produces( {"application/json"} )
  public TransitPath findOptimalRoute(
           @QueryParam( "origin" ) String originUnLocode,
           @QueryParam( "destination" ) String destinationUnLocode,
           @QueryParam( "deadline" ) String deadline ) {
    TransitPath transitPath =
     cargoRoutingService.findOptimalRoute( originUnLocode, destinationUnLocode, deadline );
    return transitPath;
  }
}
```

Реализация сервиса назначения маршрута доставки груза предоставляет интерфейс REST API, доступный в локации */optimalRoute*. Этот интерфейс принимает набор данных спецификации – исходную локацию, пункт назначения, предельный срок доставки. Затем используется класс сервисов приложения `CargoRouting` для вычисления оптимального плана маршрута на основе принятых данных спецификации. Модель предметной области (домена) в ограниченном контексте Routing представляет оптимальный план маршрута доставки в терминах `TransitPath` (это аналог плана маршрута – Itinerary) и `TransitEdge` (это аналог этапа – Leg).

В листинге 4.28 показана реализация класса `TransitPath` модели предметной области (домена).

Листинг 4.28 ❖ Реализация модели TransitPath

```
import java.util.ArrayList;
import java.util.List;
/**
 * Представление транзитного маршрута в модели предметной области (домена).
 */
```

```
public class TransitPath {
  private List<TransitEdge> transitEdges;
  public TransitPath() {
    this.transitEdges = new ArrayList<>();
  }

  public TransitPath( List<TransitEdge> transitEdges ) {
    this.transitEdges = transitEdges;
  }

  public List<TransitEdge> getTransitEdges() {
    return transitEdges;
  }

  public void setTransitEdges( List<TransitEdge> transitEdges ) {
    this.transitEdges = transitEdges;
  }

  @Override
  public String toString() {
    return "TransitPath{" + "transitEdges=" + transitEdges + '}';
  }
}
```

В листинге 4.29 показана реализация класса `TransitEdge` модели предметной области (домена).

Листинг 4.29 ❖ Реализация класса TransitEdge модели предметной области (домена)

```
package com.practicalddd.cargotracker;

import java.io.Serializable;
import java.util.Date;

/**
 * Представляет ребро (этап) пути в графе, описывающем маршрут доставки
 * груза.
 */
public class TransitEdge implements Serializable {
  private String voyageNumber;
  private String fromUnLocode;
  private String toUnLocode;
  private Date fromDate;
  private Date toDate;

  public TransitEdge() {}

  public TransitEdge( String voyageNumber, String fromUnLocode,
                      String toUnLocode, Date fromDate, Date toDate ) {
    this.voyageNumber = voyageNumber;
    this.fromUnLocode = fromUnLocode;
    this.toUnLocode = toUnLocode;
    this.fromDate = fromDate;
    this.toDate = toDate;
  }
```

```
  public String getVoyageNumber() {
    return voyageNumber;
  }

  public void setVoyageNumber( String voyageNumber ) {
    this.voyageNumber = voyageNumber;
  }

  public String getFromUnLocode() {
    return fromUnLocode;
  }

  public void setFromUnLocode( String fromUnLocode ) {
    this.fromUnLocode = fromUnLocode;
  }

  public String getToUnLocode() {
    return toUnLocode;
  }

  public void setToUnLocode( String toUnLocode ) {
    this.toUnLocode = toUnLocode;
  }

  public Date getFromDate() {
    return fromDate;
  }

  public void setFromDate( Date fromDate ) {
    this.fromDate = fromDate;
  }

  public Date getToDate() {
    return toDate;
  }

  public void setToDate( Date toDate ) {
    this.toDate = toDate;
  }

  @Override
  public String toString() {
    return "TransitEdge{" + "voyageNumber=" + voyageNumber
                          + ", fromUnLocode=" + fromUnLocode + ", toUnLocode="
                          + toUnLocode + ", fromDate=" + fromDate
                          + ", toDate=" + toDate + '}';
  }
}
```

На рис. 4.41 показана диаграмма класса для приведенной в листинге 4.29 реализации.

Следующий шаг – реализация на стороне клиента REST-сервиса для ограниченного контекста Routing (назначение маршрута доставки груза). Клиентом является класс `CargoBookingCommandService`, который отвечает за обработку команды назначения маршрута доставки груза (Assign Route to Cargo). Как части процесса обработки этой команды этому классу сервиса потребуется вызов

REST API сервиса назначения маршрута доставки для получения оптимального маршрута на основе спецификации маршрута для заказанного груза.

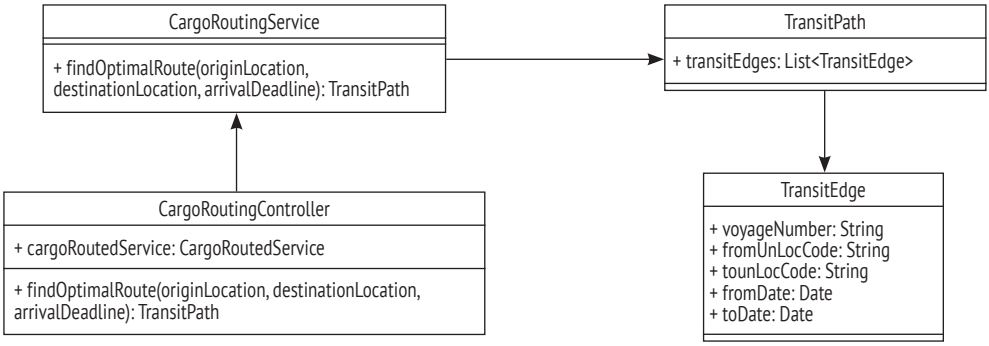

Рис. 4.41 ❖ Диаграмма класса для исходящих сервисов

Класс `CargoBookingCommandService` использует класс исходящего сервиса `ExternalCargoRoutingService` для вызова REST API сервиса назначения маршрута доставки. Класс `ExternalCargoRoutingService` также выполняет преобразование данных, передаваемых из REST API сервиса назначения маршрута доставки, в формат, распознаваемый моделью предметной области (домена) ограниченного контекста Booking (заказ груза).

В листинге 4.30 показана реализация метода `assignRouteToCargo` в классе `CargoBookingCommandService`. Этот класс сервиса команд инъецируется с зависимостью от класса `ExternalCargoRoutingService`, который обрабатывает внешний запрос (request) для вызова REST API сервиса назначения маршрута доставки и возвращает объект `CargoItinerary`, т. е. план маршрута доставки, назначаемый заказанному новому грузу.

Листинг 4.30 ❖ Зависимость от класса исходящего сервиса

```
@ApplicationScoped
public class CargoBookingCommandService {
  @Inject
  private ExternalCargoRoutingService externalCargoRoutingService;
  /**
   * Метод сервиса команд для назначения маршрута доставки груза.
   * @param routeCargoCommand
   */
  @Transactional
  public void assignRouteToCargo( RouteCargoCommand routeCargoCommand ) {
    Cargo cargo = cargoRepository.find(new BookingId(routeCargoCommand.
getCargoBookingId()));
    CargoItinerary cargoItinerary =
      externalCargoRoutingService.fetchRouteForSpecification( new RouteSpecification(
                              new Location( routeCargoCommand.getOriginLocation() ),
                              new Location( routeCargoCommand.getDestinationLocation() ),
                              routeCargoCommand.getArrivalDeadline() ) );
    cargo.assignToRoute(cargoItinerary);
```

```
    cargoRepository.store(cargo);
  }
  // Реализация всех прочих команд для ограниченного контекста Booking.
}
```

В листинге 4.31 показан класс исходящего сервиса `ExternalCargoRoutingService`. Этот класс выполняет две операции:

- инъецирует REST-клиент `ExternalCargoRoutingService` с использованием типобезопасной аннотации `@RestClient`, предоставляемой платформой MicroProfile. Этот клиент вызывает REST API сервиса назначения маршрута доставки, используя API `RestClientBuilder`, предоставляемый платформой MicroProfile;
- выполняет преобразование данных, передаваемых REST API сервиса назначения маршрута доставки (`TransitPath`, `TransitEdge`) в формат, распознаваемый моделью предметной области (домена) ограниченного контекста Booking (`CargoItinerary`, `Leg`).

Листинг 4.31 ❖ Реализация класса исходящего сервиса

```
package com.practicalddd.cargotracker.bookingms.application.internal.outboundservices.acl;

import com.practicalddd.cargotracker.TransitEdge;
import com.practicalddd.cargotracker.TransitPath;
import com.practicalddd.cargotracker.bookingms.domain.model.valueobjects.CargoItinerary;
import com.practicalddd.cargotracker.bookingms.domain.model.valueobjects.Leg;
import com.practicalddd.cargotracker.bookingms.domain.model.valueobjects.
RouteSpecification;
import com.practicalddd.cargotracker.bookingms.infrastructure.services.http.
ExternalCargoRoutingClient;

import org.eclipse.microprofile.rest.client.RestClientBuilder;
import org.eclipse.microprofile.rest.client.inject.RestClient;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
import java.util.ArrayList;
import java.util.List;

/**
 * Класс сервиса защиты от повреждений.
 */
@ApplicationScoped
public class ExternalCargoRoutingService {
  @Inject
  @RestClient // Безопасный тип платформы MicroProfile для API REST-клиента.
  private ExternalCargoRoutingClient externalCargoRoutingClient;
  /**
   * Ограниченный контекст Booking выполняет внешний вызов сервиса назначения маршрута
   * доставки из ограниченного контекста Routing для получения и назначения оптимального
   * плана маршрута доставки заказанного груза на основе спецификации маршрута.
   * @param routeSpecification
   * @return
   */
  public CargoItinerary fetchRouteForSpecification( RouteSpecification
```

```
                                                    routeSpecification ) {
    ExternalCargoRoutingClient cargoRoutingClient =
      RestClientBuilder.newBuilder().build(ExternalCargoRoutingClient.class);
    // Платформа MicroProfile: типобезопасный API REST-клиента.
    TransitPath transitPath = cargoRoutingClient.findOptimalRoute(
                routeSpecification.getOrigin().getUnLocCode(),
                routeSpecification.getDestination().getUnLocCode(),
                routeSpecification.getArrivalDeadline().toString()
                ); // Вызов API сервиса Routing с использованием этого клиента.
    List<Leg> legs = new ArrayList<Leg>(transitPath.getTransitEdges().size());
    for( TransitEdge edge : transitPath.getTransitEdges() ) {
      legs.add(toLeg(edge));
    }
    return new CargoItinerary(legs);
  }

  /**
   * Уровень защиты от повреждений: метод преобразования данных из формата модели домена
   * Routing (TransitEdges) в формат модели домена, распознаваемый ограниченным
   * контекстом Booking (Legs).
   * @param edge
   * @return
   */
  private Leg toLeg(TransitEdge edge) {
    return new Leg(
                edge.getVoyageNumber(),
                edge.getFromUnLocode(),
                edge.getToUnLocode(),
                edge.getFromDate(),
                edge.getToDate() );
  }
}
```

В листинге 4.32 показана реализация типобезопасного REST-клиента `ExternalCargoRoutingClient`. Этот клиент реализован как интерфейс с использованием аннотации `@RegisterRestClient` для соответствующей его пометки. Сигнатура метода и подробная информация о его ресурсах должна в точности соответствовать API вызываемого сервиса (в рассматриваемом здесь случае это API `optimalRoute` сервиса Routing).

Листинг 4.32 ❖ Реализация типобезопасного REST-клиента ExternalcargoRoutingClient

```
package com.practicalddd.cargotracker.bookingms.infrastructure.services.http;

import javax.ws.rs.*;

import com.practicalddd.cargotracker.TransitPath;
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;

/**
 * Типобезопасный REST-клиент для API сервиса Routing.
 */
@Path( "cargoRouting" )
```

```
@RegisterRestClient // Аннотация для регистрации как REST-клиента.
public interface ExternalCargoRoutingClient {
  // Сигнатура метода и подробная информация о его ресурсах должна в точности
  // соответствовать вызываемому сервису.
  @GET
  @Path( "/optimalRoute" )
  @Produces( {"application/json"} )
  public TransitPath findOptimalRoute(
           @QueryParam( "origin" ) String originUnLocode,
           @QueryParam( "destination" ) String destinationUnLocode,
           @QueryParam( "deadline" ) String deadline);
}
```

На рис. 4.42 показана диаграмма класса для приведенной здесь реализации.

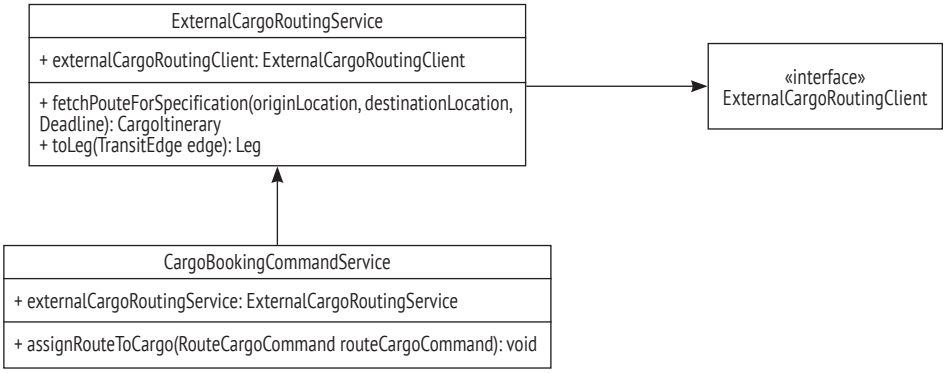

Рис. 4.42 ❖ Диаграмма класса для реализации исходящих сервисов (HTTP)

Все реализации исходящих сервисов, для которых требуется обмен данными с другими ограниченными контекстами, следуют одной и той же методике, показанной на рис. 4.43.

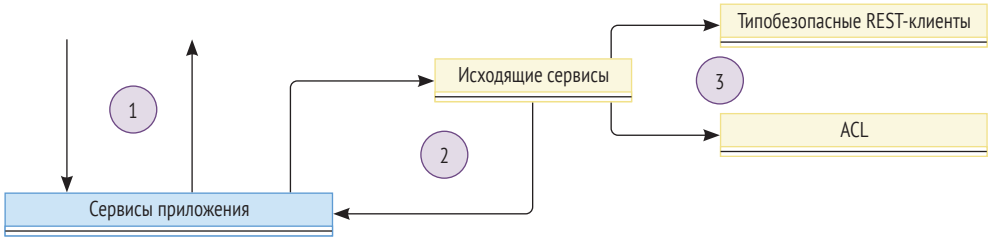

Рис. 4.43 ❖ Общая схема реализации исходящих сервисов (HTTP)

1. Классы сервисов приложения принимают команды/запросы/события.
2. Как часть процесса обработки, если требуется взаимодействие с API другого ограниченного контекста с применением REST, используется исходящий сервис.

3. Исходящий сервис использует типобезопасный REST-клиент для вызова API требуемого ограниченного контекста. Исходящий сервис также выполняет преобразование данных из формата, предоставляемого этим API другого ограниченного контекста, в формат (модель) данных, распознаваемый текущим ограниченным контекстом.

Исходящие сервисы: брокер сообщений

Последним типом исходящих сервисов, который необходимо реализовать, является механизм взаимодействий с использованием брокеров сообщений. Брокеры сообщений предоставляют необходимую физическую инфраструктуру для публикации и подписки на события предметной области (домена).

Ранее уже рассматривались некоторые классы событий (`CargoBooked`, `CargoRouted`), реализованные с использованием специализированных аннотаций. Также рассматривались и механизмы их публикации (с использованием метода `fire()`) и подписки (с использованием метода `observes()`).

Теперь рассмотрим реализацию механизма, обеспечивающего публикацию и подписку для этих событий, на основе запросов и точек обмена сервера RabbitMQ.

Следует отметить, что ни платформа Eclipse MicroProfile, ни расширения проекта Helidon MP не предоставляют функциональные возможности, обеспечивающие публикацию событий CDI Events в брокере RabbitMQ, поэтому потребуется собственная реализация этого механизма. Исходный код для этого раздела представлен в отдельном проекте (*cargo-tracker-rabbitmq-adaptor*), который предоставляет следующие функциональные возможности:

- функциональные возможности инфраструктуры (фабрики соединений для сервисов RabbitMQ, управляемые публикаторы и управляемые потребители);
- функциональные возможности для публикации сообщений AMQP (Advanced Message Queuing Protocol) для событий CDI Events в точках обмена RabbitMQ;
- функциональные возможности для потребления сообщений AMQP для событий из очередей RabbitMQ.

Здесь мы не будем подробно рассматривать полную реализацию этого проекта. Будет показана только работа с API, предоставляемыми этим проектом и позволяющими создать практический вариант использования событий CDI Events, интегрированных с точками обмена и очередями RabbitMQ. Чтобы воспользоваться этим проектом, необходимо добавить следующее описание зависимости в каждый файл зависимостей *pom.xml* нашего проекта MicroProfile:

```
<dependency>
    <groupId>com.practicalddd.cargotracker</groupId>
    <artifactId>cargo-tracker-rabbimq-adaptor</artifactId>
    <version>1.0.FINAL</version>
</dependency>
```

Первый шаг реализации механизма соединений REST API создание класса `Binder`. Класс `Binder` предназначен для следующих целей:

- связывания событий CDI Events с точками обмена и ключами Routing;
- связывания событий CDI Events с очередями (сообщений).

В листинге 4.33 показан класс RoutedEventBinder, отвечающий за связывание события CargoRouted с соответствующей точкой обмена RabbitMQ. Это расширение класса EventBinder, предоставляемое проектом адаптера. Необходимо выполнить замещение метода bindEvents(), в котором выполняются все связывания отображаемых событий CDI Events с точками обмена и очередями. Также необходимо отметить, что инициализация связывания выполняется в методе жизненного цикла, активизируемого после конструктора и предоставляемого механизмом CDI.

Листинг 4.33 ❖ Реализация класса RoutedEventBinder

```
package com.practicalddd.cargotracker.bookingms.infrastructure.brokers.rabbitmq;

import javax.annotation.PostConstruct;
import javax.enterprise.context.ApplicationScoped;
import com.practicalddd.cargotracker.rabbitmqadaptor.EventBinder; // Класс адаптера.

/**
 *  Компонент, который инициализирует события назначения маршрута
 *  для груза <-> связывания с RabbitMQ
 */
@ApplicationScoped
public class RoutedEventBinder extends EventBinder {
  /*
   * Метод связывания класса события CargoRouted с соответствующей точкой обмена
   * в брокере RabbitMQ и с соответствующим ключом Routing.
   */
  @PostConstruct // Аннотация CDI для инициализации в методе жизненного цикла этого
компонента,
                 // выполняемом после конструктора класса.
  public void bindEvents() {
    bind( CargoRoutedEvent.class )
            .toExchange( "routed.exchange" )
            .withPublisherConfirms()
           .withRoutingKey( "routed.key" );
  }
}
```

При каждой генерации события CargoRouted оно доставляется как сообщение AMQP в соответствующую точку обмена со специальным ключом Routing.

Аналогичный механизм можно применить и для реализации подписки на события. События CDI Events «связываются» с соответствующими очередями RabbitMQ, и всякий раз, когда обнаруживается нужное событие, оно доставляется как сообщение AMQP из соответствующей очереди.

Исходный код для этого подраздела содержит полную реализацию механизма инициализации событий для всех ограниченных контекстов (см. пакет com.practicalddd.cargotracker.<<имя_ограниченного_контекста>>.infrastructure.brokers.rabbitmq).

На рис. 4.44 показана диаграмма класса для приведенной здесь реализации.

На этом завершается рассмотрение реализации предметно-ориентированного проектирования в проекте Cargo Tracker как приложения на основе ми-

кросервисов с использованием платформы Eclipse MicroProfile и проекта Helidon MP, предоставляющего конкретный вариант реализации.

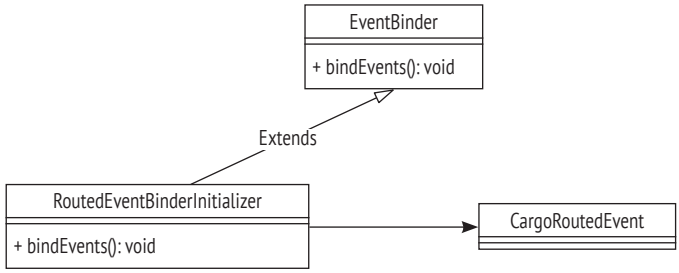

Рис. 4.44 ❖ Диаграмма класса EventBinder

Итоговый обзор реализации

Теперь мы располагаем полноценной реализацией предметно-ориентированного проектирования в приложении Cargo Tracker с различными артефактами предметно-ориентированного проектирования, реализованными с использованием соответствующих спецификаций, доступных на платформе Eclipse MicroProfile.

Общая обзорная схема этой реализации показана на рис. 4.45.

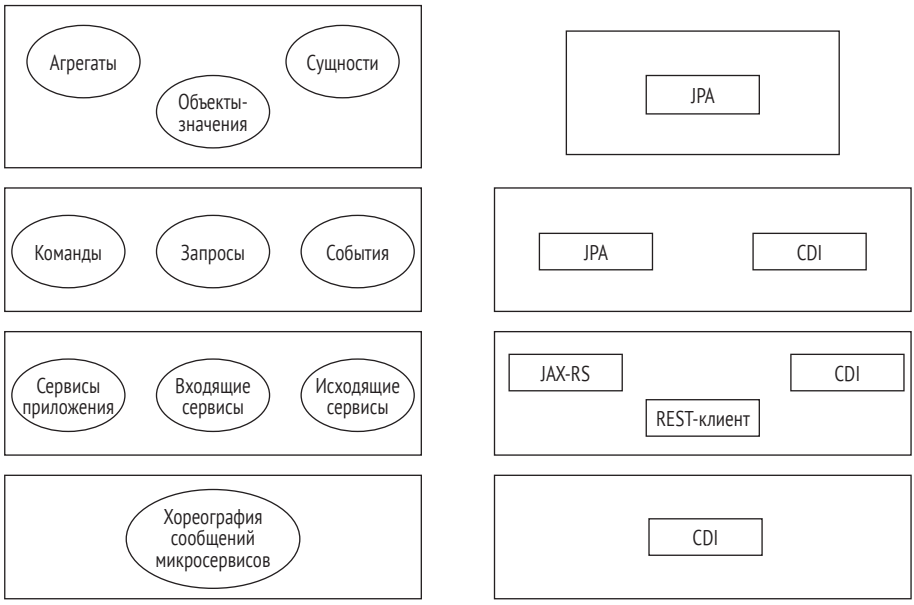

Рис. 4.45 ❖ Общая схема реализации артефактов предметно-ориентированного проектирования с использованием платформы Eclipse MicroProfile

Резюме

Краткое обобщение содержимого этой главы:

- в начале главы была приведена подробная информация о платформе Eclipse MicroProfile и о разнообразных функциональных возможностях, предоставляемых этой платформой;
- было принято решение об использовании реализации платформы Eclipse MicroProfile, представленной в проекте Helidon MP, для создания приложения Cargo Tracker на основе микросервисов;
- были рассмотрены все подробности процесса разработки и реализации различных артефактов предметно-ориентированного проектирования – сначала модели предметной области (домена), затем сервисов модели предметной области (домена) с использованием технологий и методик, доступных на платформе Eclipse MicroProfile и в проекте Helidon MP.

Глава 5

Проект Cargo Tracker: платформа Spring

Краткий обзор предыдущих глав:

- отслеживание доставки груза определено как основная область задачи, т. е. основная предметная область (домен), а приложение Cargo Tracker – как решение для этой области задачи;
- определены различные поддомены/ограниченные контексты для приложения Cargo Tracker;
- подробно описана модель предметной области (домена) для каждого определенного ограниченного контекста, в том числе были определены агрегаты, сущности, объекты-значения и правила предметной области (домена);
- определены поддерживающие сервисы предметной области (домена), необходимые для ограниченных контекстов;
- определены различные операции в ограниченных контекстах (команды, запросы, события и саги);
- описана реализация монолитной версии приложения Cargo Tracker с использованием платформы Jakarta EE и реализация версии приложения Cargo Tracker на основе микросервисов с использованием платформы Eclipse MicroProfile.

В этой главе подробно рассматривается третий вариант реализации предметно-ориентированного проектирования для приложения Cargo Tracker с использованием платформы Spring. Приложение Cargo Tracker будет снова проектироваться с применением архитектуры, основанной на микросервисах. Как и ранее, артефакты предметно-ориентированного проектирования будут отображаться в соответствующие объекты реализации, доступные на платформе Spring.

В ходе рассмотрения этого варианта реализации будут встречаться темы, которые уже подробно описывались в предыдущих главах. Повторные описания приводятся для удобства тех читателей, которых интересует только конкретная версия реализации, рассматриваемая в этой главе, а не последовательное изучение всех вариантов реализации, приведенных в этой книге.

В первую очередь необходимо сделать обзор платформы Spring.

Платформа Spring

Изначально созданная как альтернатива Java EE платформа Spring (https://spring.io/) стала ведущей рабочей средой Java для создания приложений масштаба промышленных предприятий. Широкий диапазон функциональности, предлагаемый в форме набора проектов, является активно развивающимся и охватывает практически все аспекты, требуемые для создания промышленных корпоративных приложений.

В отличие от платформ Jakarta EE и Eclipse MicroProfile, на которых существует определенный набор спецификаций, а многочисленные участники предоставляют реализации этих спецификаций, платформа Spring предлагает комплект (портфолио) конкретных проектов.

Этот комплект проектов охватывает следующие основные области:

- основные проекты (ядра) инфраструктуры, которые предоставляют основополагающий набор проектов для создания приложений на основе платформы Spring;
- проекты, специально предназначенные для облачных сред, предоставляющие функциональные возможности для создания Spring-приложений, специализированных для работы в облачной среде;
- проекты управления данными предоставляют функциональные возможности для управления любыми типами данных в приложениях на основе платформы Spring.

Отдельные проекты платформы Spring показаны на рис. 5.1.

На рис. 5.1 можно видеть, насколько обширен набор проектов, предоставляющих огромный диапазон функциональных возможностей. Напомним, что цель этой главы – реализация в приложении Cargo Tracker принципов предметно-ориентированного проектирования на основе архитектуры микросервисов. Для достижения этой цели мы будем использовать лишь небольшую подгруппу из всех доступных проектов (Spring Boot, Spring Data и Spring Cloud Stream).

Также кратко напомним требования платформы микросервисов, показанные на рис. 5.2.

Теперь необходимо кратко описать функциональные возможности перечисленных выше проектов и связать их с требованиями платформы микросервисов.

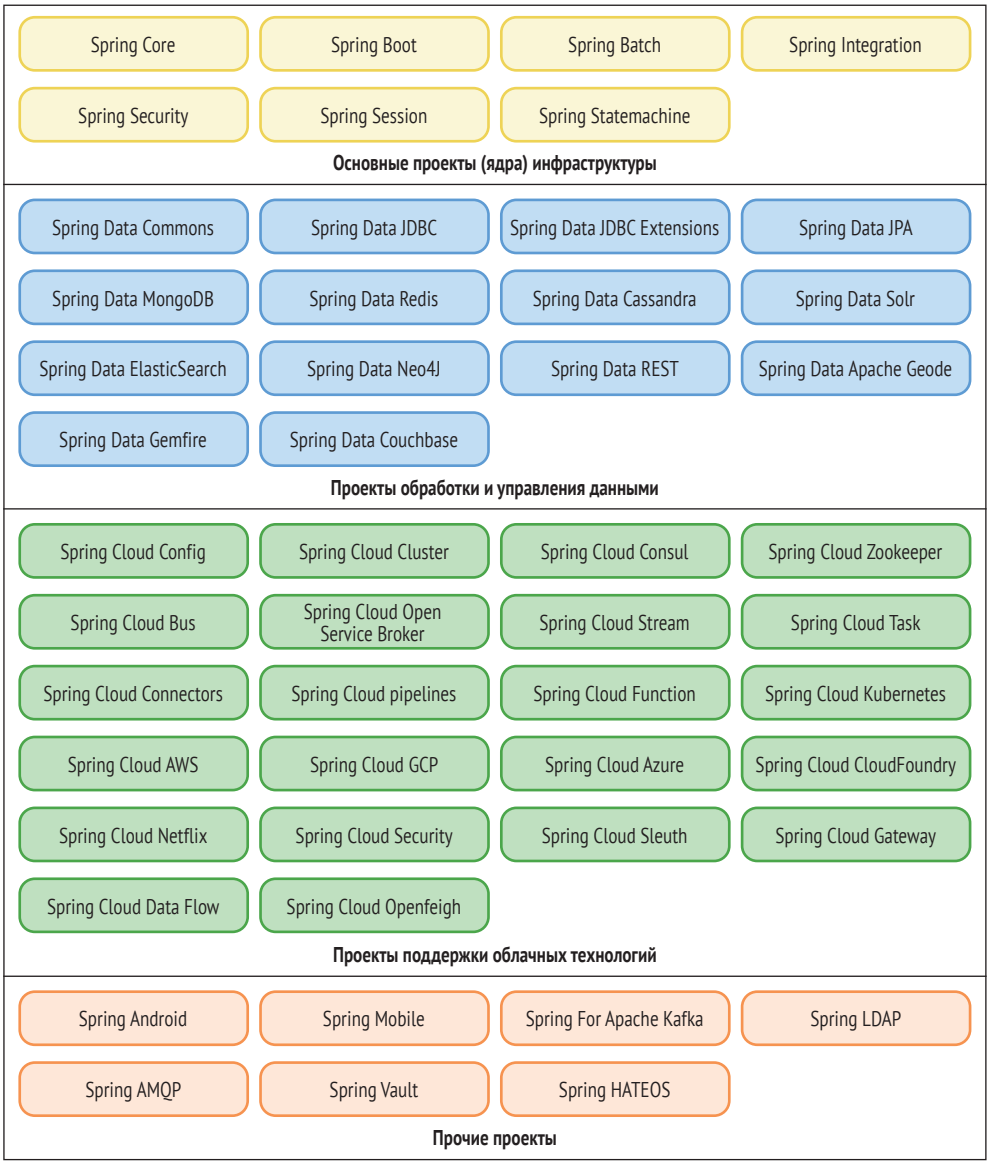

Рис. 5.1 ❖ Проекты платформы Spring

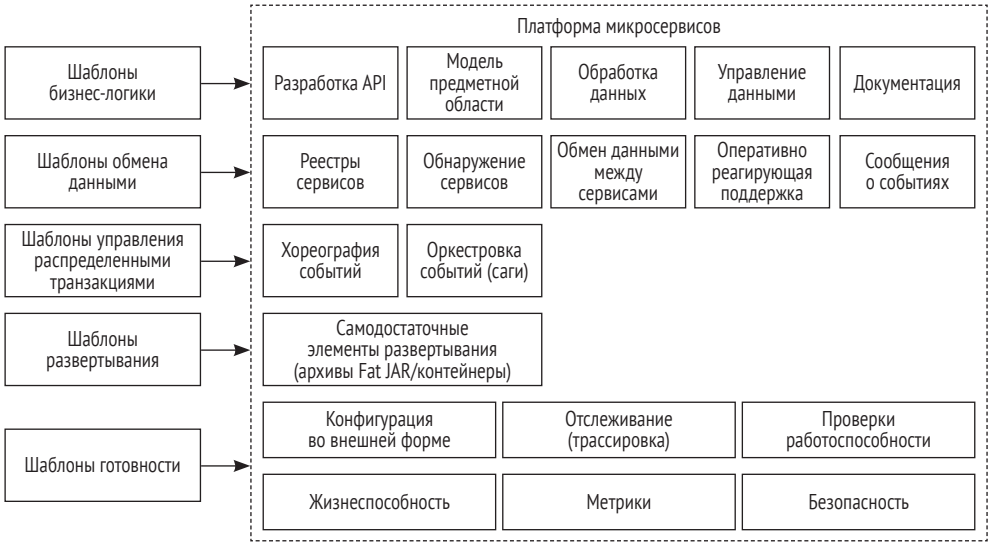

Рис. 5.2 ❖ Требования платформы микросервисов

Spring Boot: функциональные возможности

Проект Spring Boot работает как основной компонент для любого Spring-приложения на основе микросервисов. Spring Boot – в высшей степени самодостаточная платформа, которая помогает создавать микросервисы с функциями REST, обработки данных и обмена сообщениями, используя для этого единообразный тщательно отлаженный процесс разработки. Это становится возможным благодаря уровню абстракции/управления зависимостями, который в Spring Boot реализуется на базе реальных проектов, предоставляющих функциональные возможности REST, обработки данных и обмена сообщениями. Разработчик всегда стремится избежать трудностей в управлении зависимостями, а также проблем с конфигурацией, требуемой при создании приложений на основе микросервисов. Spring Boot позволяет разработчику абстрагироваться от всех этих трудностей и проблем, предлагая инструментальные комплекты для начала разработки. Эти «начальные» инструментальные комплекты предоставляют требуемый каркас приложения, позволяющий разработчикам сразу же приступить к разработке микросервисов, которые должны обеспечить API, обрабатывать данные и стать элементами архитектуры, управляемой событиями. В рассматриваемой здесь версии реализации мы отдаем предпочтение трем проектам для начала разработки, предоставляемым платформой Spring Boot (spring-boot-starter-web, spring-boot-starter-data-jpa и spring-cloud-starter-stream-rabbit).

Более подробно эти проекты будут описаны в процессе реализации примера приложения.

С точки зрения соответствия требованиям платформы микросервисов на рис. 5.3 зеленым цветом выделены элементы, которые реализуются с помощью платформы Spring Boot.

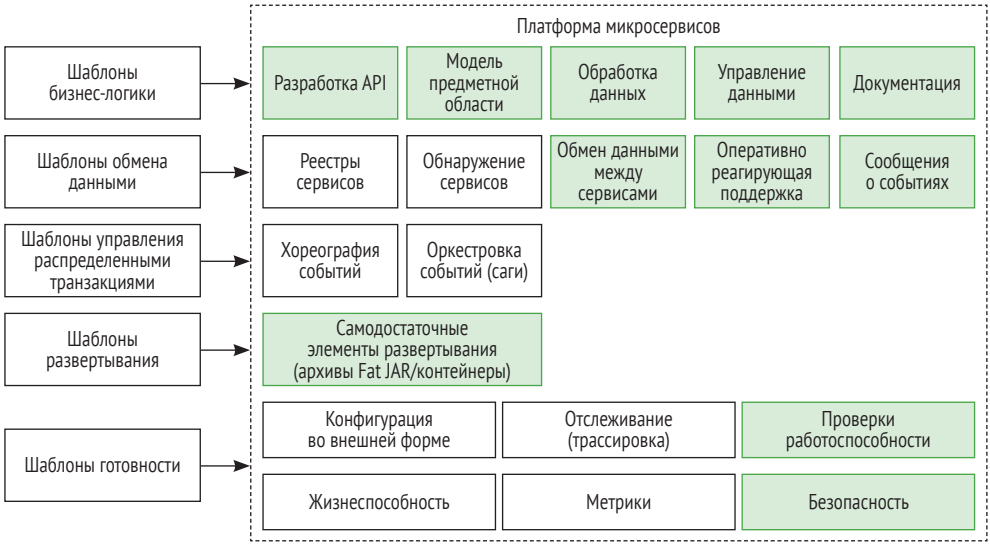

Рис. 5.3 ❖ Компоненты платформы микросервисов, предоставляемые платформой Spring Boot

Spring Cloud

Если платформа Spring Boot предоставляет основополагающие технологии для создания приложений на основе микросервисов, то платформа Spring Cloud помогает реализовать шаблоны распределенных систем, необходимые для приложений на основе микросервисов и платформы Spring Boot. Это шаблоны конкретной конфигурации, регистрации и обнаружения сервисов, обмена сообщениями, распределенной трассировки и шлюзов API. Кроме того, этот проект предоставляет более специализированные проекты для непосредственной интеграции с независимым облачным провайдером, таким как AWS/GCP/Azure.

С точки зрения соответствия требованиям платформы микросервисов на рис. 5.4 желтым цветом выделены элементы, которые реализуются с помощью платформы Spring Cloud.

Платформа Spring не предоставляет какие-либо готовые к применению «из коробки» функциональные возможности для управления распределенными транзакциями с использованием саг на основе оркестровки. Управление распределенными транзакциями в рассматриваемом здесь примере приложения будет реализовано с применением механизма хореографии событий со специализированной реализацией, использующей платформы Spring Boot и Spring Cloud Stream.

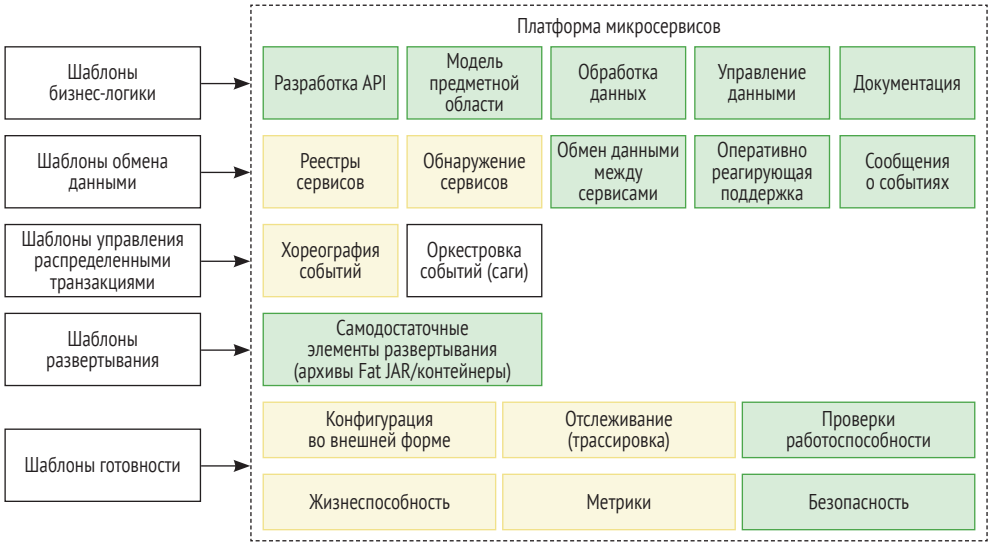

Рис. 5.4 ❖ Компоненты платформы микросервисов, предоставляемые платформой Spring Cloud

Итог краткого обзора рабочей среды Spring

Теперь у нас есть ясное представление о том, что предоставляет платформа Spring для создания приложений на основе микросервисов с применением проектов Spring Boot и Spring Cloud.

Далее мы будем рассматривать реализацию приложения Cargo Tracker с использованием этих технологий. В описании реализации могут встречаться фрагменты, повторяющие материал предыдущих глав, но это сделано для удобства тех читателей, которые интересуются только этой версией реализации.

Ограниченные контексты и платформа Spring Boot

Ограниченные контексты – это начальный пункт формирования решения для реализации предметно-ориентированного проектирования в приложении Cargo Tracker на основе микросервисов и платформы Spring. При соблюдении принципов архитектуры микросервисов каждый ограниченный контекст должен быть самодостаточным, независимо развертываемым элементом без прямых зависимостей от любого другого ограниченного контекста в рассматриваемой предметной области (области задачи).

Шаблон для разделения приложения Cargo Tracker на несколько микросервисов будет точно таким же, как и ранее (в предыдущей главе), т. е. основная

предметная область (домен) разделяется на набор бизнес-функций/поддоменов, а решением для каждого такого поддомена является отдельный ограниченный контекст.

Реализация ограниченных контекстов предполагает логическое группирование ранее определенных артефактов предметно-ориентированного проектирования в единый развертываемый артефакт. Каждый ограниченный контекст в приложении Cargo Tracker будет создаваться как приложение Spring Boot. Итоговым артефактом приложения Spring Boot является самодостаточный архивный файл fat JAR, который содержит все требуемые зависимости (например, библиотеки обеспечения доступа к данным, библиотеки REST) и соответствующую конфигурацию. Файл fat JAR также содержит встроенный веб-контейнер (в нашем примере это Tomcat) как среду времени выполнения. Это гарантия того, что нам не потребуется какой-либо внешний сервер приложений для запуска файла fat JAR. Общая схема приложения на основе Spring Boot показана на рис. 5.5.

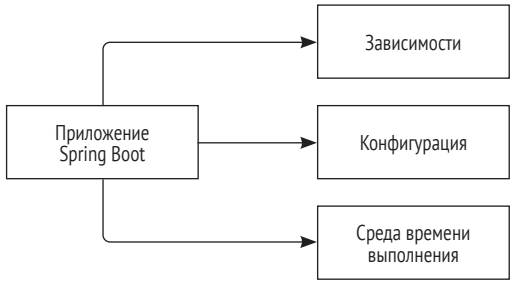

Рис. 5.5 ❖ Общая схема приложения на основе Spring Boot

В общей схеме процесса развертывания, показанного на рис. 5.6, каждый микросервис является независимым самодостаточным развертываемым элементом (файлом fat JAR).

Микросервисам потребуется хранилище данных для сохранения их состояния. В рассматриваемом здесь примере выбран принцип применения отдельной базы данных для каждого шаблона сервиса, т. е. каждый микросервис будет иметь собственное независимое хранилище данных. Для звена приложения имеется многовариантный выбор технологии и точно так же существует многовариантный выбор хранилища данных. Можно выбрать обычную реляционную базу данных (например, Oracle, MySQL, PostgreSQL), базу данных типа NoSQL (например, MongoDB, Cassandra) или даже хранилище данных в оперативной памяти (например, Redis). Выбор в основном зависит от требований к масштабируемости и от типа конкретного варианта использования микросервиса, который будет работать с хранилищем данных. В рассматриваемом здесь примере в качестве хранилища данных выбрана база данных MySQL. Архитектура развертывания показана на рис. 5.6.

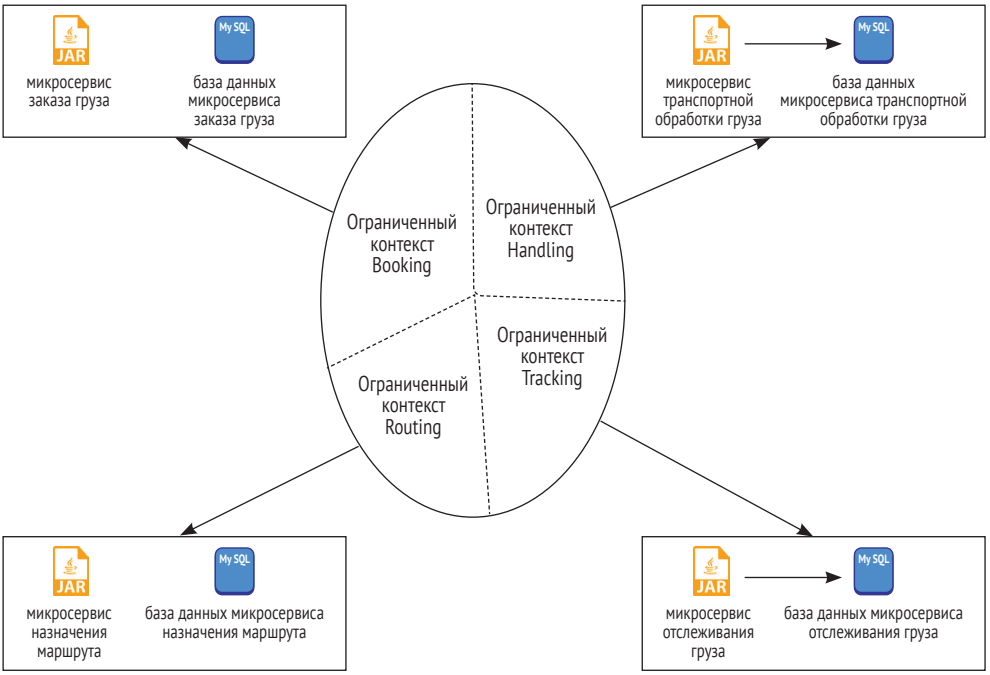

Рис. 5.6 ❖ Архитектура развертывания микросервисов на основе платформы Spring Boot

Ограниченные контексты: формирование пакетов

Чтобы начать формирование пакетов, необходим первый шаг – создание обычного приложения Spring Boot. Мы будем использовать инструментальное средство Spring Initializr (https://start.spring.io/) на основе браузера, которое значительно упрощает процесс создания приложений Spring Boot. На рис. 5.7 показано создание микросервиса заказа груза (Booking) с использованием инструментального средства Initializr.

Рассматриваемый здесь пример проекта был создан со следующими характеристиками:

- группа (Group) – com.practicalddd.cargotracker;
- артефакт (Artifact) – bookingms;
- зависимости (Dependencies) – Spring Web Starter, Spring Data JPA и Spring Cloud Stream.

Необходимо щелкнуть по пиктограмме Generate Project (Генерация проекта). Сгенерируется ZIP-файл, содержащий Spring Boot приложение Booking со всеми зависимостями и конфигурацией, которые сделаны доступными.

Основной класс приложения Spring Boot определен с использованием аннотации `@SpringBootApplication`. Он содержит открытый статический метод `void main` и точку входа в приложение Spring Boot.

Класс `BookingmsApplication` – это основной класс в Spring Boot приложении Booking. В листинге 5.1 показана реализация класса `BookingmsApplication`.

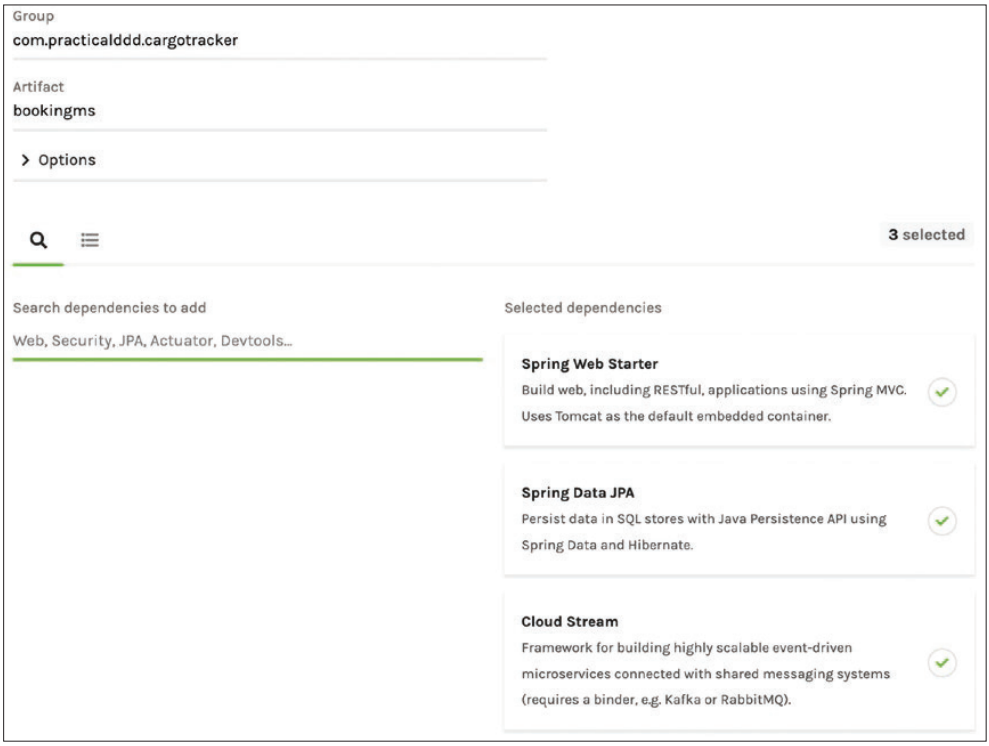

Рис. 5.7 ❖ Инструментальное средство Spring Initializr для создания каркаса проектов Spring Boot с зависимостями

Листинг 5.1 ❖ Класс BookingmsApplication

```
package com.practicalddd.cargotracker.bookingms;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication  // Аннотация, обозначающая основной класс приложения.
public class BookingmsApplication {
  public static void main(String[] args) {
    SpringApplication.run(BookingmsApplication.class, args);
  }
}
```

Результатом создания проекта является JAR-файл *bookingms.jar*. Его запуск как обычного простого JAR-файла с помощью команды `java -jar bookingms.jar` начинает работу приложения Spring Boot.

Ограниченные контексты: структура пакета

После принятия решения о процедуре формирования пакетов следующий шаг – определение структуры пакета для каждого ограниченного контекста, т. е. переход к логическому группированию различных артефактов предметно-ориентированного проектирования в единый развертываемый артефакт. Та-

кое логическое группирование предполагает определение структуры пакета, в котором размещаются различные артефакты предметно-ориентированного проектирования для получения общего решения для ограниченного контекста.

Структура верхнего уровня пакета для любого ограниченного контекста в рассматриваемом здесь примере показана на рис. 5.8.

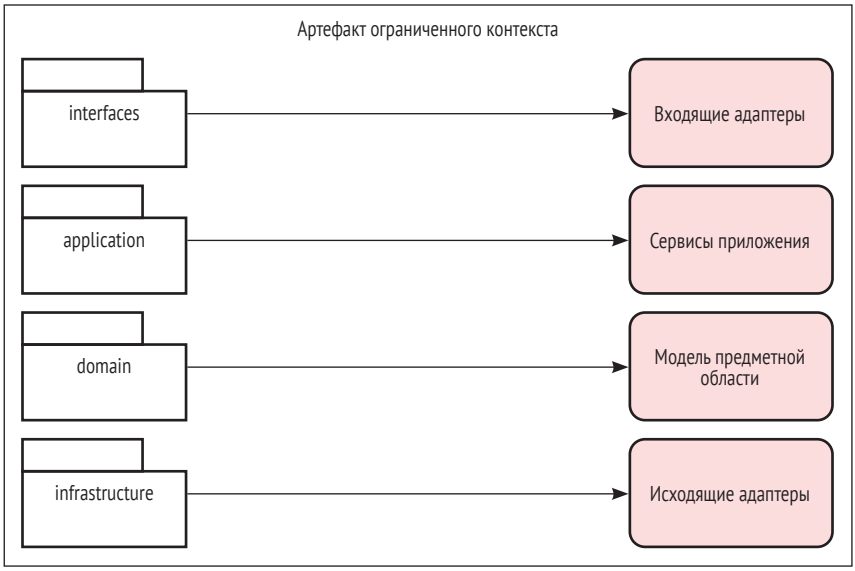

Рис. 5.8 ❖ Структура пакета для ограниченных контекстов

Рассмотрим структуру этого пакета более подробно.

Пример структуры пакета ограниченного контекста Booking в приложении Spring Boot показан на рис. 5.9 с основным классом приложения `BookingmsApplication`.

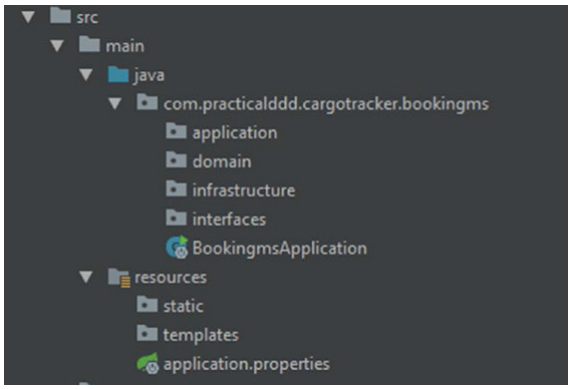

Рис. 5.9 ❖ Структура пакета для ограниченного контекста Booking с использованием Spring Boot

Рассмотрим подробнее структуру этого пакета.

Пакет interfaces

Эта часть пакета включает все входящие интерфейсы для рассматриваемого ограниченного контекста, классифицированные по протоколам обмена данными. Главная цель интерфейсов – согласование протокола от имени модели предметной области (домена) (например, REST API, WebSocket, FTP, какой-либо специализированный протокол).

В рассматриваемом здесь примере ограниченный контекст Booking (заказ груза) предоставляет REST API для передачи ему запросов на изменение состояния, т. е. команд (например, команд Book Cargo, Assign Route to Cargo). Кроме того, ограниченный контекст Booking предоставляет REST API для передачи ему запросов на получение состояния, т. е. собственно запросов (Queries) (например, Retrieve Cargo Booking Details, List all Cargos). Эти интерфейсы сгруппированы в пакете *rest*.

К интерфейсам также относятся обработчики событий, которые подписываются на различные события, генерируемые другими ограниченными контекстами. Все обработчики событий собраны в пакете *eventhandlers*. В дополнение к этим двум пакетам пакет интерфейсов также содержит пакет *transform*, который используется для преобразования входных данных ресурсов API и событий в соответствующий формат (модель) команд и запросов, требуемый моделью предметной области (домена).

Так как в рассматриваемом здесь примере необходима поддержка REST, событий и преобразования данных, соответствующая этим требованиям структура пакета показана на рис. 5.10.

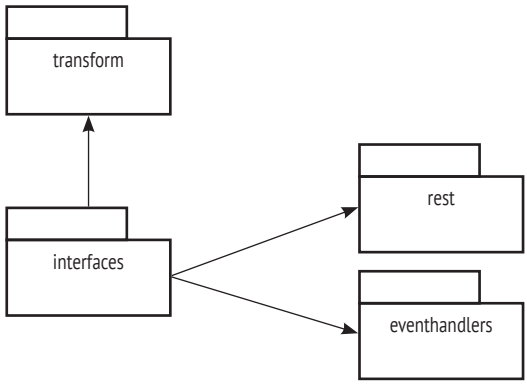

Рис. 5.10 ❖ Структура пакета interfaces

Пакет application

Сервисы приложения функционируют как внешний компонент модели предметной области (домена) ограниченного контекста. Они предоставляют сервисы внешнего («фасадного») уровня для регулирования и распределения

команд и запросов в модель предметной области (домена) более низкого уровня. Сервисы приложения также определяют размещение исходящих вызовов, направляемых в другие ограниченные контексты, как часть процесса обработки команд и запросов.

Таким образом, сервисы приложения выполняют следующие функции:

- участие в регулировании и распределении команд и запросов;
- вызов компонентов инфраструктуры, когда это необходимо, как часть процесса обработки команд и запросов;
- предоставление централизованных компонентов (например, ведение журналов, обеспечение безопасности, метрики) для модели предметной области (домена) более низкого уровня;
- выполнение вызовов, направленных в другие ограниченные контексты.

Структура пакета application показана на рис. 5.11.

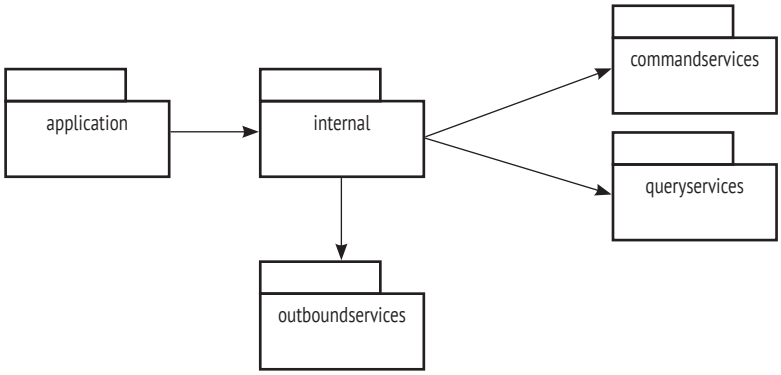

Рис. 5.11 ❖ Структура пакета сервисов приложения

Пакет domain

Пакет domain содержит модель предметной области (домена) ограниченного контекста. Его можно назвать сердцем модели предметной области (домена) ограниченного контекста, содержащим реализацию основной бизнес-логики.

Ниже перечислены основные классы этого ограниченного контекста:

- агрегаты;
- сущности;
- объекты-значения;
- команды;
- события.

Структура пакета domain показана на рис. 5.12.

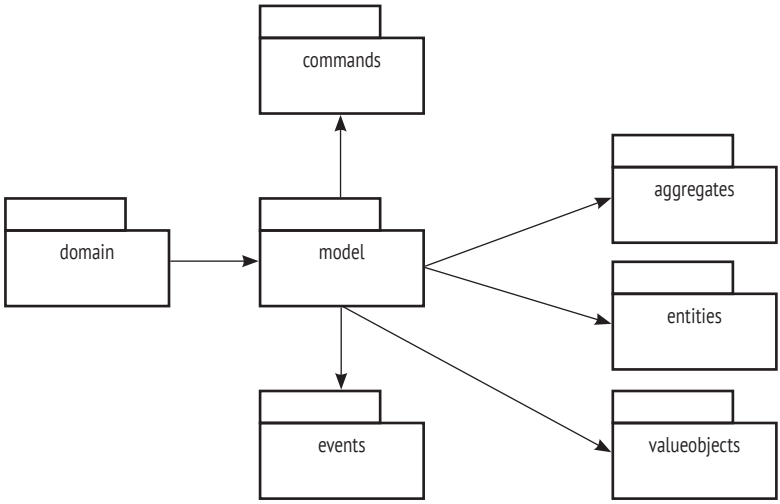

Рис. 5.12 ❖ Структура пакета для модели предметной области

Пакет infrastructure

Пакет infrastructure предназначен для достижения следующих трех основных целей:

- когда ограниченный контекст принимает запрос на операцию, связанную с его состоянием (изменение состояния, извлечение состояния), ему необходим нижележащий репозиторий для выполнения этой операции. В рассматриваемом здесь примере репозиторием является экземпляр базы данных MySQL. Пакет инфраструктуры содержит все необходимые компоненты, которые требуются ограниченному контексту для обмена данными с нижележащим репозиторием. Как часть рассматриваемого здесь проекта мы будем использовать JPA или JDBC для реализации этих компонентов;
- когда ограниченный контекст должен передать сообщение о событии изменения состояния, ему необходима нижележащая инфраструктура событий для публикации события изменения состояния. В рассматриваемой здесь реализации будет использоваться брокер сообщений (RabbitMQ) как нижележащая инфраструктура событий. Пакет infrastructure содержит все необходимые компоненты, которые требуются ограниченному контексту для обмена данными с нижележащим брокером сообщений;
- последний элемент, включенный в инфраструктурный уровень, – любой тип конфигурации, специализированной для Spring Boot.

Структура пакета infrastructure показана на рис. 5.13.

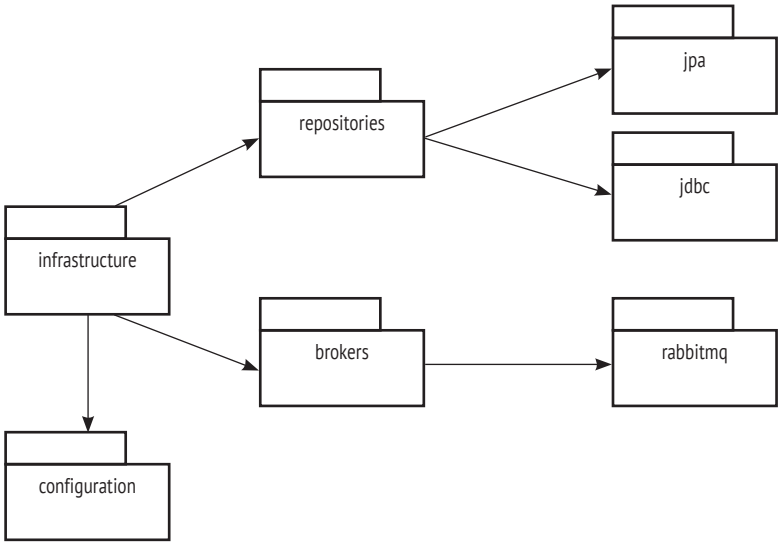

Рис. 5.13 ❖ Структура пакета, содержащего компоненты инфраструктуры

Полная обобщенная схема структуры всего пакета в целом для любого ограниченного контекста показана на рис. 5.14.

На этом завершается реализация ограниченных контекстов для рассматриваемого здесь приложения Cargo Tracker на основе микросервисов. Каждый из ограниченных контекстов реализован как приложение Spring Boot, а в качестве артефакта принят файл fat JAR. Ограниченные контексты аккуратно разделены в структуре пакета по модулям с четко разграниченными функциями.

Теперь можно перейти к следующему этапу реализации приложения Cargo Tracker.

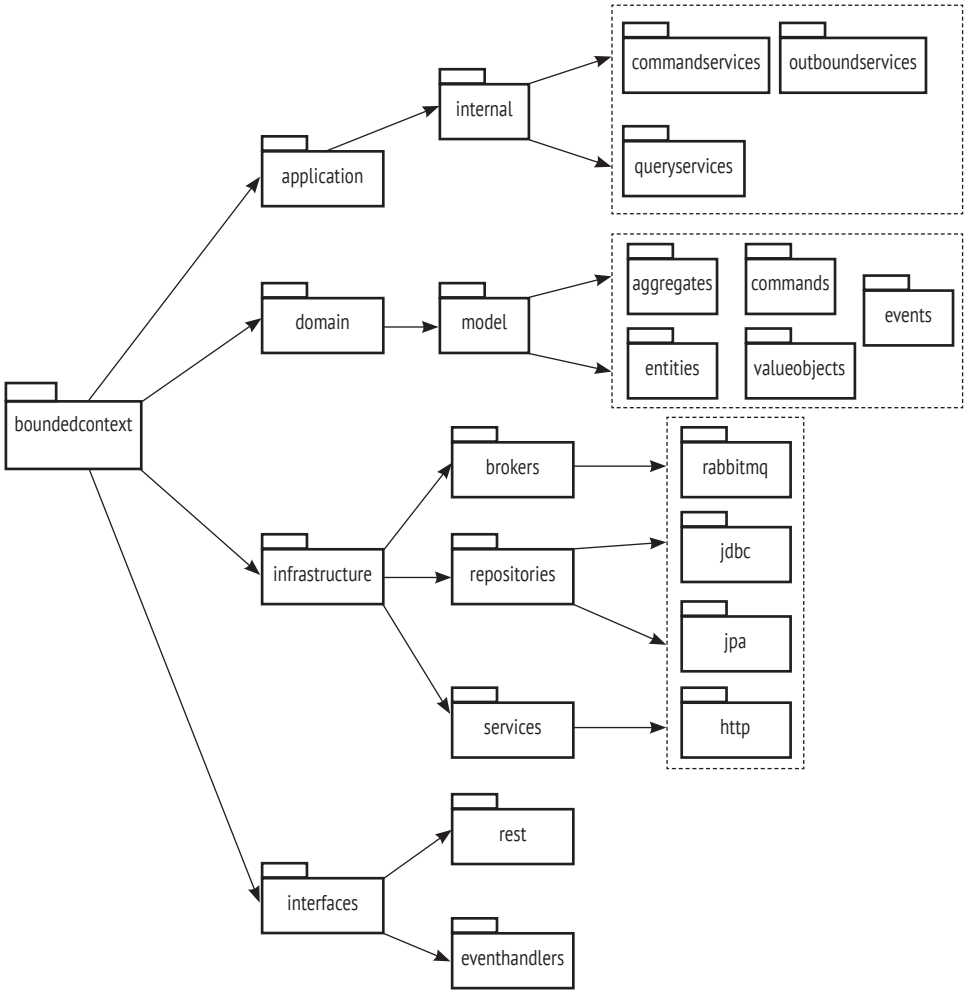

Рис. 5.14 ❖ Общая структура пакета для любого ограниченного контекста

Реализация приложения Cargo Tracker

В этом разделе подробно рассматривается реализация Cargo Tracker как приложения на основе микросервисов с использованием предметно-ориентированного проектирования и платформ Spring Boot/Spring Cloud. Как уже было отмечено ранее, в некоторых подразделах повторяется материал из предыдущей главы, но всегда полезно вспомнить основные концепции предметно-ориентированного проектирования.

Обобщенный обзор логического группирования различных артефактов предметно-ориентированного проектирования показан на рис. 5.15. Здесь можно видеть, что необходима реализация следующих двух групп артефактов:

- модели предметной области (домена), которая будет содержать ранее определенный основной домен/бизнес-логику;
- сервисов предметной области (домена), содержащих сервисы поддержки для рассматриваемой здесь модели основного домена.

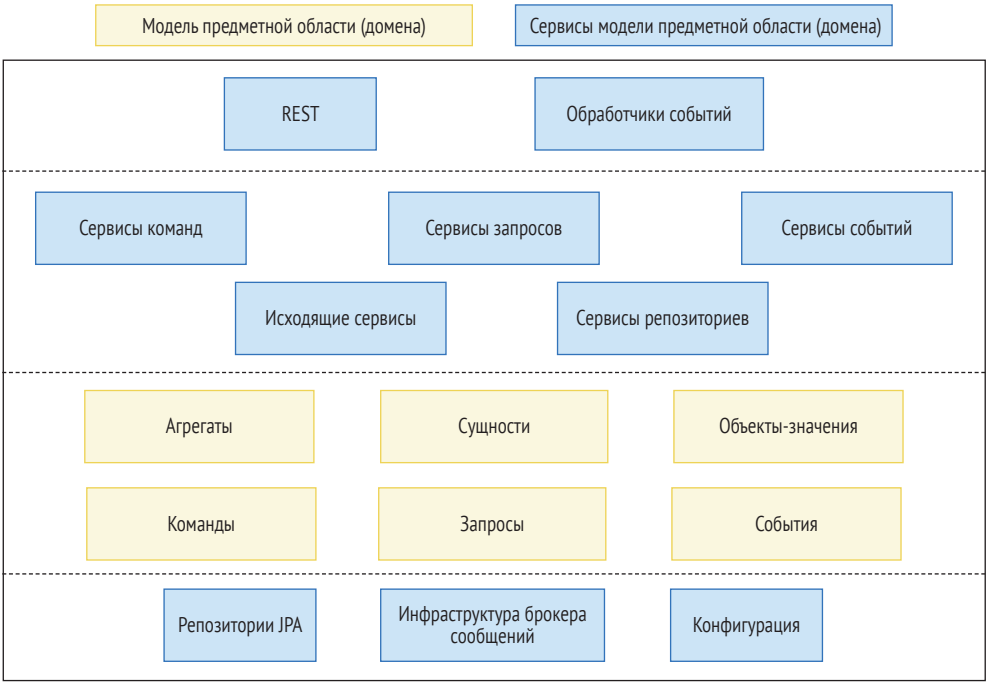

Рис. 5.15 ❖ Логическое группирование артефактов предметно-ориентированного проектирования

В терминах действительной реализации модели предметной области (домена) эти артефакты преобразуются в различные объекты-значения, команды и запросы конкретного ограниченного контекста/микросервисов.

В терминах действительной реализации сервисов модели предметной области (домена) эти артефакты преобразуются в интерфейсы, сервисы приложения и инфраструктуру, которые требует модель предметной области (домена) ограниченного контекста/микросервисов.

Вернемся к приложению Cargo Tracker. На рис. 5.16 показано решение с использованием микросервисов в терминах различных ограниченных контекстов и поддерживаемых ими операций. Схема решения содержит различные команды, которые будет обрабатывать каждый ограниченный контекст, запросы, которые будет обслуживать каждый ограниченный контекст, и события, которые будет публиковать и на которые будет подписываться каждый ограниченный контекст. Каждый микросервис представляет собой отдельный независимо развертываемый артефакт с собственным хранилищем данных.

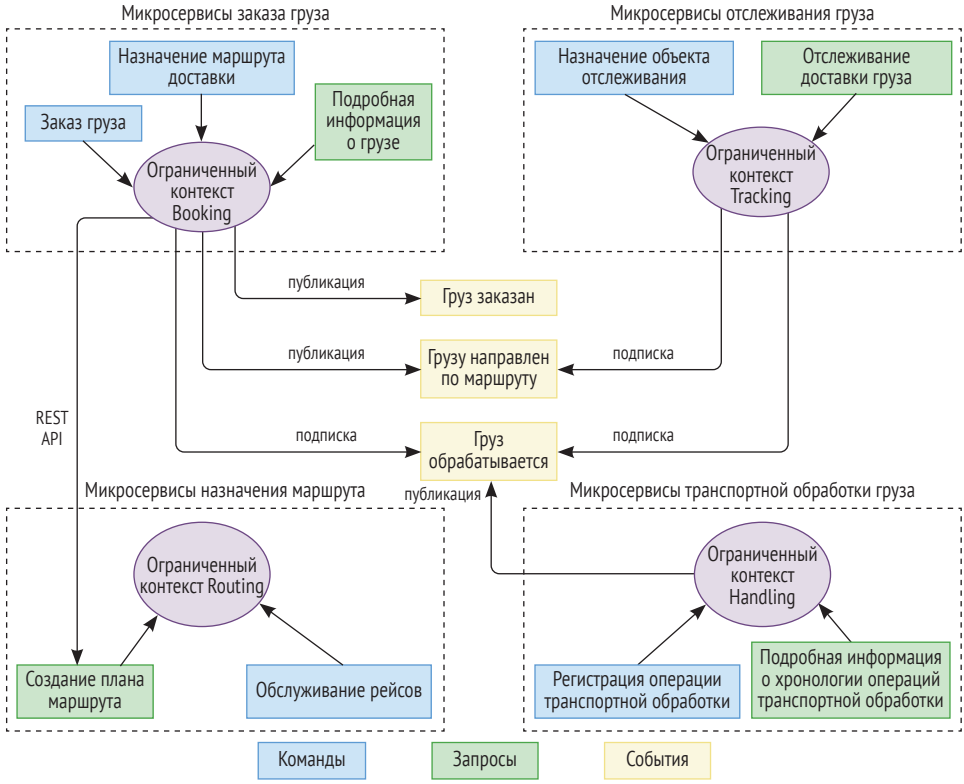

Рис. 5.16 ❖ Решение на основе микросервисов для приложения Cargo Tracker

ПРИМЕЧАНИЕ Некоторые примеры исходного кода реализации могут содержать только фрагменты или сокращенные варианты, помогающие понять общие концепции реализации. Исходный код для текущей главы содержит полную реализацию этих концепций.

Модель предметной области (домена): реализация

Принятая модель предметной области (домена) является центральным функциональным элементом рассматриваемого ограниченного контекста и, как было отмечено выше, имеет набор артефактов, связанных с этой моделью. Реа-

лизация этих артефактов выполняется с помощью инструментальных средств, предоставляемых платформой Spring Boot.

Ниже перечислены артефакты модели предметной области (домена), которые необходимо реализовать:

- модель основного домена – агрегаты, сущности и объекты-значения;
- операции модели предметной области (домена) – команды, запросы и события.

Рассмотрим подробнее каждый из этих артефактов и определим, какие инструментальные средства, предоставляемые платформой Spring Boot, подходят для их реализации.

Модель основного домена: реализация

Реализация основной предметной области (домена) для любого ограниченного контекста включает идентификацию тех артефактов, которые ясно выражают бизнес-цели этого ограниченного контекста. На самом верхнем уровне это включает идентификацию и реализацию агрегатов, сущностей и объектов-значений.

Агрегаты, сущности и объекты-значения

Агрегаты являются главными элементами модели предметной области (домена). Напомним, что мы определили четыре агрегата, по одному в каждом ограниченном контексте, как показано на рис. 5.17.

Рис. 5.17 ❖ Агрегаты в ограниченных контекстах/микросервисах

Реализация агрегата включает следующие этапы:

- реализацию класса агрегата;
- обеспечение полноценности предметной области (домена) с помощью бизнес-атрибутов;
- реализацию сущностей и объектов-значений.

Реализация класса агрегата

Поскольку выбрано использование СУРБД MySQL в качестве хранилища данных для каждого ограниченного контекста, будет применяться интерфейс JPA (Java Persistent API) по соответствующей спецификации Java EE. JPA предоставляет стандартный способ определения и реализации сущностей и объектов-значений, которые взаимодействуют с хранилищами данных SQL более низкого уровня.

Интеграция JPA: Spring Data JPA

Платформа Spring Boot обеспечивает поддержку JPA, используя для этого проект Spring Data JPA (https://spring.io/projects/spring-data-jpa), который предоставляет изощренный, но простой механизм для реализации репозиториев на основе JPA. Платформа Spring Boot предоставляет начальный проект (*spring-boot-starter-data-jpa*), который автоматически конфигурирует набор значимых параметров, принимаемых по умолчанию (например, реализацию Hibernate JPA, создание пула соединений Tomcat) для Spring Data JPA.

Зависимость для начального проекта Spring Data JPA автоматически добавляется при его конфигурировании как зависимость в проекте Initializr. В дополнение к этому необходимо добавить драйвер библиотеки MySQL Java, чтобы обеспечить возможность соединения с экземплярами базы данных MySQL. Это делается в следующем файле:

`pom.xml`

В листинге 5.2 показаны изменения, которые необходимо внести в файл зависимостей *pom.xml*, генерируемый проектом Spring Initializr.

Листинг 5.2 ❖ Обеспечение зависимостей в файле pom.xml

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
```

`application.properties`

В дополнение к этим зависимостям также необходимо сконфигурировать свойства соединения для каждого экземпляра базы данных MySQL. Для этого предназначен файл *application.properties*, предоставляемый приложением Spring Boot. В листинге 5.3 показаны свойства конфигурации, которые необходимо добавить. Потребуется замена значений конкретных параметров для экземпляров MySQL, если это необходимо.

Листинг 5.3 ❖ Конфигурация соединений с базой данных MySQL

```
spring.datasource.url=jdbc:mysql://<<имя_машины>>:<<номер_порта>>/<<имя_экземпляра_БД_MySQL>>
spring.datasource.username=<<UserID_экземпляра_БД_MySQL>>
spring.datasource.password==<<Пароль_для_экземпляра_БД_MySQL>>
```

Приведенных в листинге 5.3 параметров вполне достаточно для настройки и реализации JPA в рассматриваемом здесь приложении Spring Boot. Как уже было отмечено ранее, проект Spring Data JPA конфигурирует набор значимых параметров, принимаемых по умолчанию, что позволяет начать разработку с минимальными усилиями. Если не определены какие-либо другие условия, то все агрегаты во всех ограниченных контекстах рассматриваемого здесь примера реализуют один и тот же механизм.

Каждый из определенных в рассматриваемом здесь примере классов корневых агрегатов реализован как объект JPA. JPA не предоставляет специальные аннотации для обозначения конкретного класса как корневого агрегата, поэтому принимается обычное обозначение старого объекта Java (POJO) и используется стандартная аннотация JPA `@Entity`. Рассмотрим пример ограниченного контекста Booking (заказ груза) с корневым агрегатом Cargo. В листинге 5.4 показан небольшой фрагмент кода, необходимый для объекта JPA.

Листинг 5.4 ❖ Корневой агрегат Cargo, реализованный как объект JPA

```
package com.practicalddd.cargotracker.bookingms.domain.model.aggregates;
import javax.persistence.*;
@Entity    // Маркер объекта JPA.
public class Cargo {
}
```

Для каждого объекта JPA требуется идентификатор. Для реализации идентификатора агрегата выбран технический идентификатор (первичный ключ) агрегата Cargo, выведенный из последовательности MySQL. В дополнение к техническому идентификатору также определяется бизнес-ключ (Business Key).

Бизнес-ключ явно выражает бизнес-цель идентификатора агрегата, т. е. идентификатор заказа нового груза, следовательно, это ключ, предъявляемый внешним потребителям (пользователям) модели предметной области (подробности об этом будут изложены несколько позже). С другой стороны, технический ключ – это исключительно внутреннее представление идентификатора агрегата, удобное для обслуживания отношений в ограниченном контексте между агрегатами и зависящими от них объектами (см. сущности и объекты-значения в следующих подразделах).

Продолжая рассматривать пример агрегата Cargo в ограниченном контексте Booking, мы прямо сейчас добавляем технический ключ и бизнес-ключ в реализацию класса агрегата.

Это показано в листинге 5.5. Аннотация `@Id` определяет первичный ключ агрегата Cargo. Для обозначения бизнес-ключа нет специализированной аннотации, поэтому он реализован как обычный старый объект Java (POJO) `BookingId` и включен в состав агрегата с помощью аннотации `@Embedded`, предоставляемой JPA.

Листинг 5.5 ❖ Реализация идентификатора корневого агрегата Cargo

```
package com.practicalddd.cargotracker.bookingms.domain.model.aggregates;
import javax.persistence.*;
@Entity
public class Cargo {
```

```
  @Id // Аннотация идентификатора, предоставляемая JPA.
  // На основе последовательности, сгенерированной MySQL.
  @GeneratedValue( strategy = GenerationType.IDENTITY )
  private Long id;
  @Embedded // Аннотация, позволяющая использовать бизнес-объекты вместо простейших типов.
  private BookingId bookingId; // Бизнес-идентификатор.
}
```

В листинге 5.6 показана реализация класса бизнес-ключа `BookingId`.

Листинг 5.6 ❖ Реализация класса бизнес-ключа BookingId для агрегата Cargo

```
package com.practicalddd.cargotracker.bookingms.domain.model.aggregates;
import javax.persistence.*;
import java.io.Serializable;
/**
 * Идентификатор бизнес-ключ для агрегата Cargo.
 */
@Embeddable
public class BookingId implements Serializable {
  @Column( name="booking_id" )
  private String bookingId;
  public BookingId() {}
  public BookingId( String bookingId ) { this.bookingId = bookingId; }
  public String getBookingId() { return this.bookingId; }
}
```

Теперь у нас есть основной каркас реализации агрегата (в данном случае агрегата Cargo) с использованием JPA. Для других агрегатов используется тот же механизм реализации, за исключением ограниченного контекста Handling Activity (операция транспортной обработки груза). Так как это журнал событий, принято решение реализовать только один ключ для этого агрегата, т. е. `ActivityId` (идентификатор операции).

На рис. 5.18 показана общая схема первоначальной реализации всех агрегатов.

Рис. 5.18 ❖ Первоначальная реализация агрегатов

Обеспечение полноценности предметной области: бизнес-атрибуты

Подготовив первоначальные реализации агрегатов, можно переходить к следующему этапу – обеспечению полноценности предметной области (домена). Агрегат любого ограниченного контекста должен обладать способностью выражения бизнес-языка своего ограниченного контекста. По существу, в точной технической терминологии это означает, что агрегат не должен быть неполноценным (anemic), т. е. не должен содержать только методы установки и считывания значений (getter/setter methods).

Неполноценный агрегат (anemic aggregate) противоречит основному принципу предметно-ориентированного проектирования, поскольку, в сущности, это означало бы, что бизнес-язык выражается на нескольких уровнях приложения. В свою очередь это ведет к тому, что некоторая часть программного обеспечения становится чрезвычайно неудобной в эксплуатации и сопровождении при долговременном использовании.

Но как реализовать агрегат полноценной предметной области (домена)? Краткий ответ: с помощью бизнес-атрибутов и бизнес-методов. В этом подразделе мы сосредоточимся на бизнес-атрибутах, а бизнес-методы будут рассматриваться как часть реализации операции модели предметной области (домена).

Бизнес-атрибуты агрегата содержат состояние агрегата в форме атрибутов, описываемых с использованием бизнес-терминов, а не технической терминологии.

Рассмотрим подробнее бизнес-атрибуты на примере агрегата Cargo.

После преобразования состояния в бизнес-концепции агрегат Cargo имеет следующие атрибуты:

- исходную локацию (Origin Location) груза;
- количество заказанного (Booking Amount) груза;
- спецификацию маршрута (Route specification) (исходную локацию, пункт назначения, конечный срок прибытия в пункт назначения);
- план маршрута доставки (Itinerary), который назначается для груза на основе спецификации маршрута. План маршрута доставки состоит из нескольких этапов (Legs), по которым груз может перемещаться, чтобы дойти до пункта назначения;
- ход процесса доставки (Delivery Progress) груза присваивается вместе со спецификацией и планом маршрута доставки. Ход процесса доставки предоставляет подробную информацию о состоянии маршрута, состоянии транспорта, текущем рейсе доставки груза, последней известной локации груза, следующей запланированной операции и последней операции, которая производилась с грузом.

На рис. 5.19 показан агрегат Cargo и его отношения с зависящими от него объектами. Следует особо отметить возможность явного представления агрегата Cargo исключительно в бизнес-терминах.

JPA предоставляет набор аннотаций (`@Embedded`, `@Embeddable`), которые помогают реализовать класс этого агрегата с использованием бизнес-объектов.

В листинге 5.7 показан пример агрегата Cargo со всеми его зависимостями, смоделированными как бизнес-объекты.

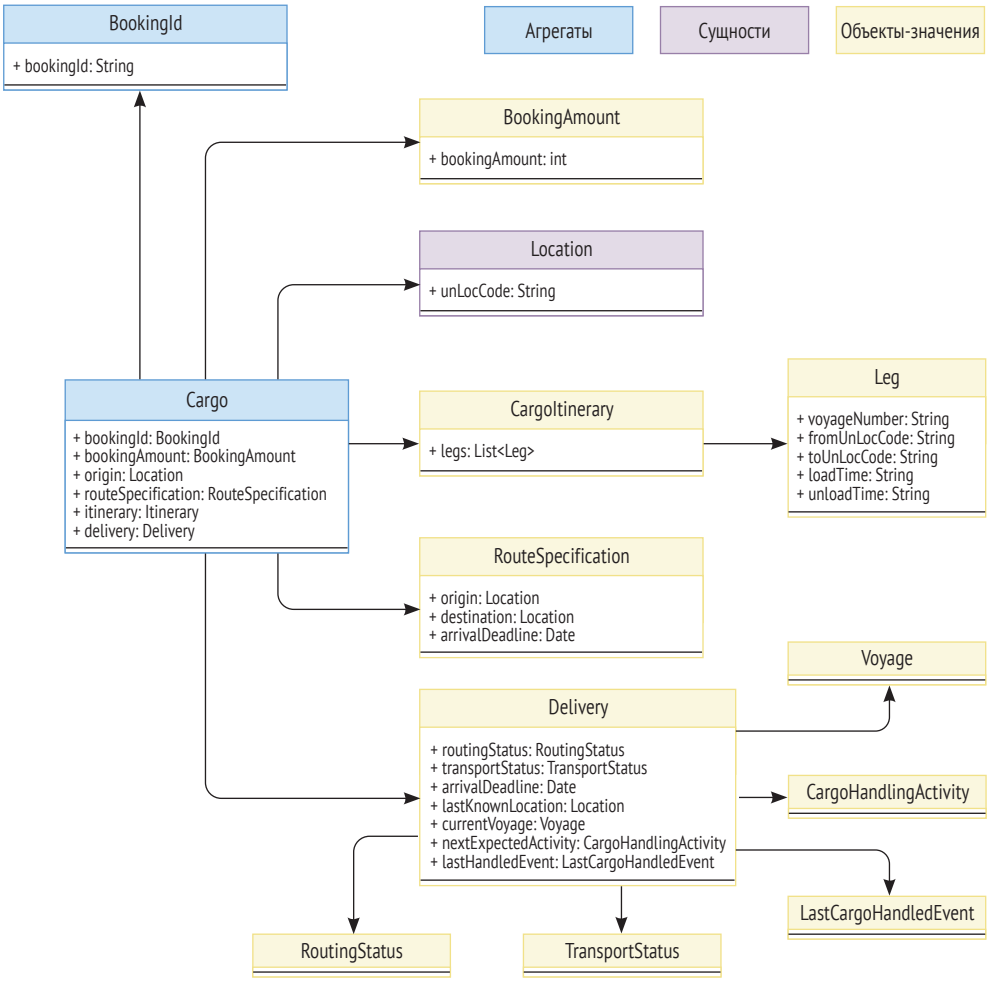

Рис. 5.19 ❖ Агрегат Cargo и зависимые от него объекты

Листинг 5.7 ❖ Зависимые бизнес-объекты для корневого агрегата Cargo

```
package com.practicalddd.cargotracker.bookingms.domain.model.aggregates;

import javax.persistence.*;

import com.practicalddd.cargotracker.bookingms.domain.model.entities.*;
import com.practicalddd.cargotracker.bookingms.domain.model.valueobjects.*;

@Entity
public class Cargo {
  @Id
  @GeneratedValue( strategy = GenerationType.IDENTITY )
  private Long id;
  @Embedded
  private BookingId bookingId; // Идентификатор агрегата.
  @Embedded
```

```
  private BookingAmount bookingAmount; // Количество заказанного груза.
  @Embedded
  private Location origin; // Исходная локация груза.
  @Embedded
  private RouteSpecification routeSpecification; // Спецификация маршрута доставки груза.
@Embedded
  private CargoItinerary itinerary; // План маршрута доставки груза.
  @Embedded
  private Delivery delivery; // Проверки хода процесса доставки груза в соответствии
                             // с реальной спецификацией и планом маршрута доставки груза.
}
```

Зависимые классы для агрегата моделируются либо как объекты-сущности, либо как объекты-значения. Напомним, что объекты-сущности в ограниченном контексте обладают собственной идентичностью, но всегда существуют только в корневом агрегате, т. е. не могут существовать независимо. Кроме того, объекты-сущности никогда не изменяются на протяжении всего жизненного цикла агрегата. С другой стороны, объекты-значения не обладают собственной идентичностью и легко заменяются в любом экземпляре агрегата.

Продолжая рассматривать пример агрегата Cargo, получаем следующие объекты:

- исходную локацию (Origin) груза как объект-сущность (Location). Этот объект не может измениться в любом экземпляре агрегата Cargo, следовательно, моделируется как объект-сущность;
- количество заказанного груза (Booking Amount), спецификацию маршрута доставки (Route Specification), план маршрута доставки (Cargo Itinerary), назначенный для конкретного груза, и процесс доставки (Delivery of the Cargo) как объекты-значения. Эти объекты являются заменяемыми в любом экземпляре агрегата Cargo, следовательно, моделируются как объекты-значения.

Рассмотрим подробнее конкретные ситуации (сценарии) и обоснуем, почему эти объекты определены как объекты-значения, так как это весьма важное решение при моделировании предметной области (домена):

- при новом заказе груза создается новая спецификация маршрута доставки и пустой план маршрута доставки, но отсутствует ход процесса доставки;
- после назначения плана маршрута доставки груза пустой план маршрута доставки заменяется назначенным планом маршрута доставки;
- по мере перемещения груза по нескольким портам, являющимся частью плана маршрута доставки, ход процесса доставки груза обновляется и заменяется в корневом агрегате;
- если заказчик изменяет пункт назначения груза или предельный срок доставки, то изменяется спецификация маршрута, назначается новый план маршрута, пересчитывается процесс доставки (Delivery), а также изменяется количество заказанного груза.

Все упомянутые выше объекты являются заменяемыми, следовательно, моделируются как объекты-значения. Это простое практическое правило моделирования сущностей и объектов-значений в агрегате.

Реализация объектов-сущностей и объектов-значений

Объекты-сущности и объекты-значения реализуются как встраиваемые (embeddable) объекты JPA с использованием доступной аннотации `@Embeddable`. Затем они включаются в агрегат с использованием аннотации `@Embedded`.

В листинге 5.8 показан механизм включения любых объектов в класс агрегата.

Рассмотрим подробнее реализацию объектов-сущностей и объектов-значений агрегата Cargo.

В листинге 5.8 показана реализация объекта-сущности `Location`. Обратите внимание на имя пакета, в котором собраны сущности (`model.entities`).

Листинг 5.8 ❖ Реализация класса сущности Location

```
package com.practicalddd.cargotracker.bookingms.domain.model.entities;
import javax.persistence.Column;
import javax.persistence.Embeddable;
/**
 * Класс сущности Location, представленный уникальным 5-значным кодом локации ООН.
 */
@Embeddable
public class Location {
  @Column( name = "origin_id" )
  private String unLocCode;
  public Location() {}
  public Location( String unLocCode ) { this.unLocCode = unLocCode; }
  public void setUnLocCode( String unLocCode ) { this.unLocCode = unLocCode; }
  public String getUnLocCode() { return this.unLocCode; }
}
```

В листинге 5.9 показан пример реализации объекта-значения `BookingAmount`. Обратите внимание на имя пакета, в котором собраны объекты-значения (`model.valueobjects`).

Листинг 5.9 ❖ Реализация объекта-значения BookingAmount

```
package com.practicalddd.cargotracker.bookingms.domain.model.valueobjects;
import javax.persistence.Column;
import javax.persistence.Embeddable;
/**
 * Представление в модели домена объекта-значения BookingAmount для нового груза.
 * Содержит количество заказанного груза.
 */
@Embeddable
public class BookingAmount {
  @Column( name = "booking_amount", unique = true, updatable= false )
  private Integer bookingAmount;
  public BookingAmount() {}
  public BookingAmount( Integer bookingAmount ) {
    this.bookingAmount = bookingAmount;
  }
  public void setBookingAmount( Integer bookingAmount ) {
    this.bookingAmount = bookingAmount;
```

```
  }
  public Integer getBookingAmount() { return this.bookingAmount; }
}
```

В листинге 5.10 показан пример реализации объекта-значения `RouteSpecification`.

Листинг 5.10 ❖ Реализация объекта-значения RouteSpecification

```
package com.practicalddd.cargotracker.bookingms.domain.model.valueobjects;
import com.practicalddd.cargotracker.bookingms.domain.model.entities.Location;

import javax.persistence.*;
import javax.validation.constraints.NotNull;
import java.util.Date;
/**
 * Спецификация маршрута доставки заказанного груза.
 */
@Embeddable
public class RouteSpecification {
  private static final long serialVersionUID = 1L;
  @Embedded
  @AttributeOverride( name = "unLocCode", column = @Column(name = "spec_origin_id") )
  private Location origin;
  @Embedded
  @AttributeOverride( name = "unLocCode", column = @Column(name = "spec_destination_id") )
  private Location destination;
  @Temporal( TemporalType.DATE )
  @Column(name = "spec_arrival_deadline")
  @NotNull
  private Date arrivalDeadline;
  public RouteSpecification() {}
  /**
   * @param origin origin location
   * @param destination destination location
   * @param arrivalDeadline arrival deadline
   */
  public RouteSpecification( Location origin, Location destination, Date arrivalDeadline )
{
    this.origin = origin;
    this.destination = destination;
    this.arrivalDeadline = (Date)arrivalDeadline.clone();
  }

  public Location getOrigin() {
    return origin;
  }

  public Location getDestination() {
    return destination;
  }
```

```
  public Date getArrivalDeadline() {
    return new Date( arrivalDeadline.getTime() );
  }
}
```

Все прочие объекты-значения (`CargoItinerary` и `Delivery`) реализуются аналогичным образом с использованием аннотации `@Embeddable` и включаются в агрегат Cargo с помощью аннотации `@Embedded`.

> **ПРИМЕЧАНИЕ** Полная реализация содержится в файлах исходного кода для этой главы.

На рис. 5.20, 5.21 и 5.22 показаны упрощенные диаграммы классов для других агрегатов (HandlingActivity, Voyage, TrackingActivity).

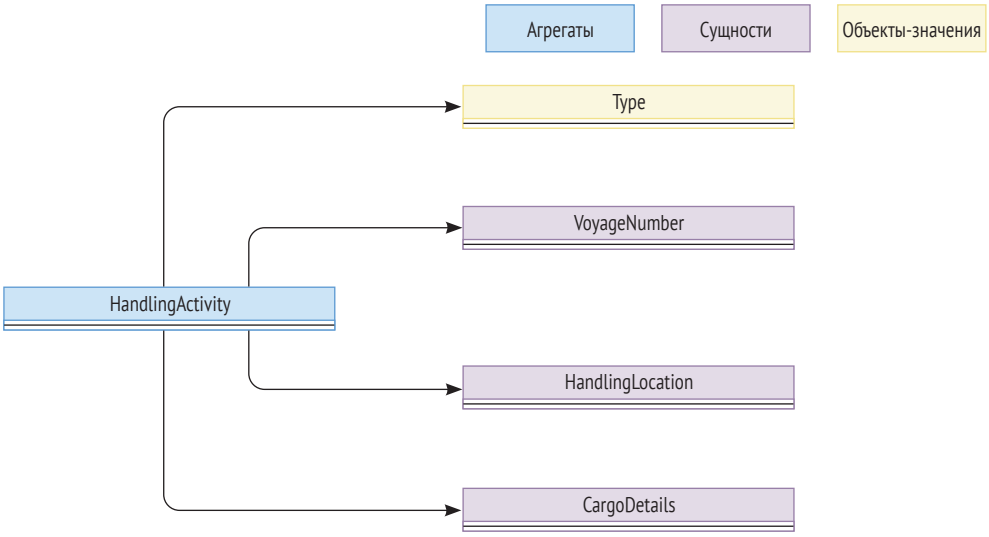

Рис. 5.20 ❖ Класс HandlingActivity и зависимые от него объекты

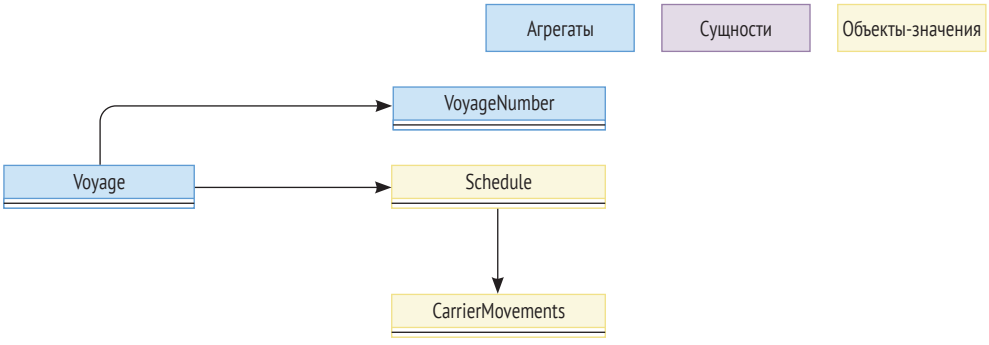

Рис. 5.21 ❖ Класс Voyage и зависимые от него объекты

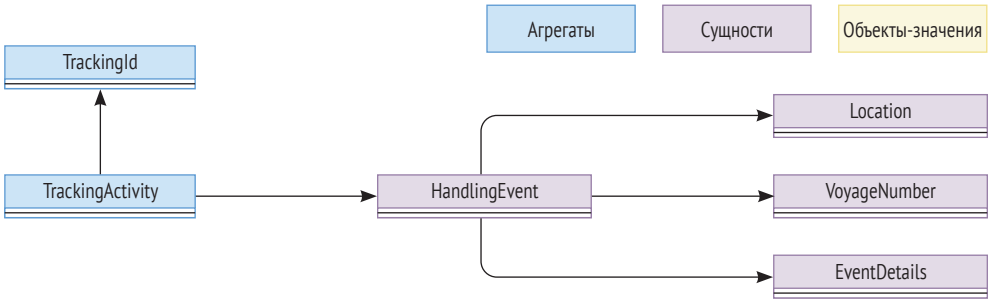

Рис. 5.22 ❖ Класс TrackingActivity и зависимые от него объекты

На этом завершается рассмотрение реализации модели основной предметной области (домена). В следующем разделе будет рассматриваться реализация операций модели предметной области (домена).

> **ПРИМЕЧАНИЕ** Исходный код, прилагаемый к этой книге, содержит реализацию модели основной предметной области (домена), демонстрируемую с использованием разделения пакетов. Чтобы получить более подробное представление о типах объектов в модели предметной области (домена), вы можете ознакомиться с исходным кодом в репозитории github.com/practicalddd.

Операции модели предметной области (домена)

Операции модели предметной области (домена) в ограниченном контексте выполняют любой тип действий, связанных с состоянием агрегата в этом ограниченном контексте. К таким операциям относятся входящие операции (команды и запросы) и исходящие операции (события).

Команды

Команды отвечают за изменение состояния агрегата в ограниченном контексте.

Реализация команд в ограниченном контексте подразумевает выполнение следующих шагов:

- идентификацию/реализацию команд;
- идентификацию/реализацию обработчиков команд для обработки команд.

Идентификация команд

Идентификация команд охватывает любую операцию, которая влияет на состояние агрегата. Например, ограниченный контекст команд заказа груза содержит следующие операции или команды:

- заказ груза;
- назначение маршрута доставки груза.

Обе эти операции в конечном итоге изменяют состояние агрегата Cargo в соответствующем ограниченном контексте, следовательно, идентифицируются как команды.

Реализация команд

Реализация идентифицированных команд на платформе Spring Boot выполняется с использованием обычных старых объектов Java (POJO). В листинге 5.11 показана реализация класса `BookCargoCommand` для команды заказа груза.

Листинг 5.11 ❖ Реализация класса BookCargoCommand

```
package com.practicalddd.cargotracker.bookingms.domain.model.commands;

import java.util.Date;

/**
 * Класс команды заказа груза.
 */
public class BookCargoCommand {
  private String bookingId;
  private int bookingAmount;
  private String originLocation;
  private String destLocation;
  private Date destArrivalDeadline;
  public BookCargoCommand() {}
  public BookCargoCommand( int bookingAmount, String originLocation, String destLocation,
                           Date destArrivalDeadline) {
    this.bookingAmount = bookingAmount;
    this.originLocation = originLocation;
    this.destLocation = destLocation;
    this.destArrivalDeadline = destArrivalDeadline;
  }

  public void setBookingId( String bookingId ){ this.bookingId = bookingId; }

  public String getBookingId() { return this.bookingId; }

  public void setBookingAmount( int bookingAmount ) {
    this.bookingAmount = bookingAmount;
  }

  public int getBookingAmount() {
    return this.bookingAmount;
  }

  public String getOriginLocation() { return originLocation; }

  public void setOriginLocation( String originLocation ) {
    this.originLocation = originLocation;
  }

  public String getDestLocation() { return destLocation; }

  public void setDestLocation( String destLocation ) {
    this.destLocation = destLocation;
  }

  public Date getDestArrivalDeadline() { return destArrivalDeadline; }

  public void setDestArrivalDeadline( Date destArrivalDeadline ) {
    this.destArrivalDeadline = destArrivalDeadline;
  }
}
```

Идентификация обработчиков команд

Для каждой команды будет предусмотрен соответствующий обработчик команды (Command Handler). Цель обработчика команды – обработать введенную команду и установить заданное состояние агрегата. Обработчики команд – это единственная часть модели предметной области (домена), где устанавливается состояние агрегата. Это строгое правило, которое необходимо соблюдать, чтобы правильно реализовать полноценную модель предметной области (домена).

Реализация обработчиков команд

Так как платформа Spring не предоставляет готовые к прямому использованию функциональные возможности для реализации обработчиков команд, методикой реализации в рассматриваемом примере будет простая идентификация подпрограмм в агрегатах. Эти подпрограммы можно обозначить как обработчики команд (Command Handlers). Для первой (по порядку выполнения) команды заказа груза (Book Cargo) конструктор соответствующего агрегата идентифицируется как обработчик этой команды. Для второй команды назначения маршрута доставки груза (Route Cargo) создается новая подпрограмма `assignToRoute()` как обработчик этой команды.

В листинге 5.12 показан фрагмент кода конструктора агрегата Cargo. Конструктор принимает команду `BookCargoCommand` как входной параметр и устанавливает соответствующее состояние этого агрегата.

Листинг 5.12 ❖ Обработчик команды BookCargoCommand

```
/**
 * Конструктор как обработчик команды нового заказа груза.
 */
public Cargo(BookCargoCommand bookCargoCommand) {
  this.bookingId = new BookingId( bookCargoCommand.getBookingId() );
  this.routeSpecification = new RouteSpecification(
                new Location( bookCargoCommand.getOriginLocation() ),
                new Location( bookCargoCommand.getDestLocation() ),
                bookCargoCommand.getDestArrivalDeadline() );
  this.origin = routeSpecification.getOrigin();
  this.itinerary = CargoItinerary.EMPTY_ITINERARY; // Пустой план маршрута доставки,
                                                   // так как грузу еще не назначен маршрут.
  this.bookingAmount = bookingAmount;
  this.delivery = Delivery.derivedFrom( this.routeSpecification, this.itinerary,
                                        LastCargoHandledEvent.EMPTY );
}
```

В листинге 5.13 показан фрагмент кода для обработчика команды `assignToRoute()`. В качестве входного параметра он принимает класс `RouteCargoCommand` и устанавливает состояние соответствующего агрегата.

Листинг 5.13 ❖ Обработчик команды RouteCargoCommand

```
/**
 * Обработчик команды RouteCargoCommand. Устанавливает состояние агрегата Cargo и
 * регистрирует событие назначения маршрута доставки груза.
 * @param routeCargoCommand
```

```
 */
public void assignToRoute( RouteCargoCommand routeCargoCommand ) {
  this.itinerary = routeCargoCommand.getCargoItinerary();
  // Синхронная обработка согласованности внутри агрегата Cargo.
  this.delivery = delivery.updateOnRouting(this.routeSpecification, this.itinerary);
}
```

Подводя итог, отметим, что обработчики команд играют весьма важную роль в управлении состоянием агрегата в ограниченном контексте. В действительности вызовы обработчиков команд выполняются через сервисы приложения, которые будут рассматриваться в одном из следующих подразделов.

На рис. 5.23 показана диаграмма класса для реализации этого обработчика команды.

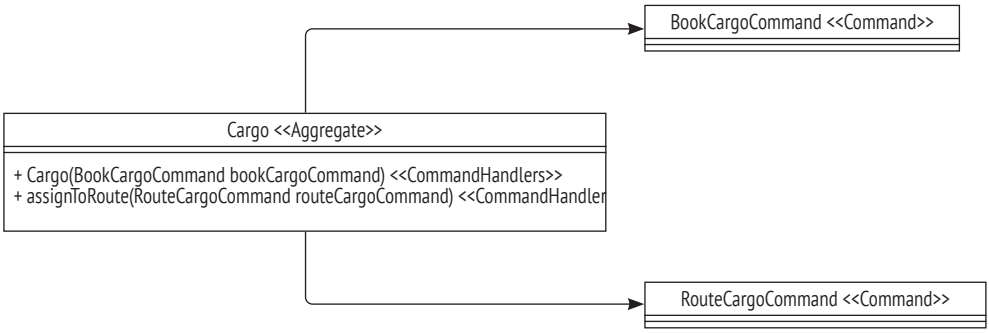

Рис. 5.23 ❖ Диаграмма класса для реализации обработчика команды

На этом завершается рассмотрение реализации команд в модели предметной области (домена). Теперь мы переходим к реализации запросов.

Запросы

Запросы (queries) в ограниченном контексте отвечают за представление состояния агрегата этого ограниченного контекста внешним потребителям (пользователям).

Для реализации запросов в рассматриваемом примере используются именованные запросы JPA (JPA Named Queries), т. е. запросы, которые могут быть определены в агрегате для извлечения его состояния в разнообразных формах. В листинге 5.14 приведен фрагмент кода из класса агрегата Cargo, в котором определяются необходимые запросы, доступные внешним потребителям. В рассматриваемом примере это три запроса: найти все грузы (Find All Cargos), найти груз по его идентификатору заказа (Find Cargo by its Booking Identifier) и вывести список идентификаторов заказа для всех грузов (Booking Identifiers for all Cargos).

Листинг 5.14 ❖ Именованные запросы в корневом агрегате Cargo

```
@NamedQueries( {
  @NamedQuery( name = "Cargo.findAll", query = "Select c from Cargo c" ),
  @NamedQuery( name = "Cargo.findByBookingId",
```

```
              query = "Select c from Cargo c where c.bookingId = :bookingId" ),
  @NamedQuery( name = "Cargo.findAllBookingIds",
              query = "Select c.bookingId from Cargo c" ) } )
public class Cargo {}
```

Запросы играют важную роль в представлении состояния агрегата в ограниченном контексте. В действительности вызовы и выполнение запросов происходит через сервисы приложения и классы репозиториев, которые будут рассматриваться в одном из следующих подразделов.

На этом заканчивается описание реализации запросов в модели предметной области (домена). Теперь перейдем к рассмотрению реализации событий.

События

Событие (event) в ограниченном контексте – это любая операция, которая публикует изменения состояния агрегата ограниченного контекста как событие. Поскольку команды изменяют состояние агрегата, вполне обоснованно можно предположить, что действие любой команды в ограниченном контексте приведет к возникновению соответствующего события. Подписчиками этих событий могут быть другие ограниченные контексты внутри той же предметной области (домена) или ограниченные контексты, принадлежащие каким-либо другим внешним предметным областям (доменам).

События предметной области (домена) играют центральную роль в архитектуре микросервисов, поэтому весьма важно реализовать их надежно и корректно. Распределенная природа архитектуры микросервисов диктует использование событий через механизм хореографии (choreography mechanism) для сопровождения состояния и сохранения логической целостности и согласованности транзакций между различными ограниченными контекстами приложения на основе микросервисов.

На рис. 5.24 показаны примеры событий, передаваемых между различными ограниченными контекстами в приложении Cargo Tracker.

Рассмотрим потоки событий более подробно на конкретном примере решения бизнес-задачи.

Когда для груза назначается маршрут доставки, это означает, что теперь груз можно отслеживать, для чего в свою очередь требуется присваивание идентификатора отслеживания (Tracking Identifier) этому грузу. Назначение маршрута доставки груза выполняется в ограниченном контексте Booking (заказ груза), в то время как присваивание идентификатора отслеживания происходит в ограниченном контексте Tracking (отслеживание доставки груза). При монолитном подходе к решению этой задачи процесс назначения маршрута доставки груза и присваивания идентификатора отслеживания этого груза происходит совместно, так как можно использовать один и тот же контекст транзакций для нескольких ограниченных контекстов благодаря совместно используемой объединенной модели для процессов, сред времени выполнения и хранилищ данных.

Но в архитектуре микросервисов невозможно достичь результата тем же способом, так как в этой архитектуре нет совместно используемых элементов. При назначении маршрута доставки груза только ограниченный контекст

Booking отвечает за достоверность отображения нового назначенного маршрута в состоянии агрегата Cargo. Ограниченный контекст Tracking должен узнать об этом изменении состояния, чтобы получить возможность присваивания идентификатора отслеживания для полноценного завершения этой конкретной бизнес-задачи. Именно здесь проявляется важная роль событий предметной области (домена) и механизма хореографии событий. Если ограниченный контекст Booking может сгенерировать событие, соответствующее факту назначения маршрута агрегату Cargo, то ограниченный контекст Tracking может подписаться на это конкретное событие и при его возникновении присвоить идентификатор отслеживания для полноценного завершения этой конкретной бизнес-задачи. Механизм генерации событий и доставки событий в различные ограниченные контексты для завершения конкретных бизнес-задач представляет собой шаблон хореографии событий (event choreography pattern).

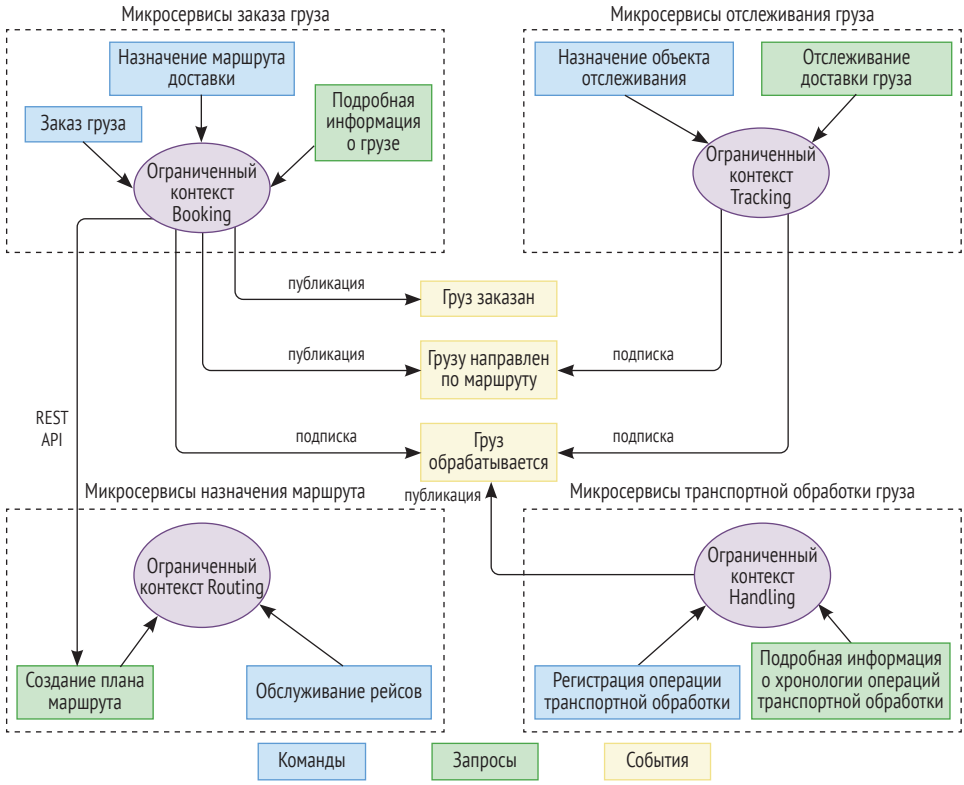

Рис. 5.24 ❖ Поток событий в архитектуре микросервисов

Можно выделить следующие четыре этапа реализации надежно функционирующей архитектуры хореографии, управляемой событиями:

- регистрацию событий предметной области (домена), которые необходимо генерировать в ограниченном контексте;
- генерацию событий предметной области (домена), которые необходимо публиковать из ограниченного контекста;

- публикацию событий, сгенерированных в ограниченном контексте;
- подписку на события, которые были опубликованы другими ограниченными контекстами.

С учетом сложности этой архитектуры процесс реализации также разделяется на несколько этапов:

- регистрация событий предметной области (домена) реализуется агрегатом;
- генерация и публикация событий реализуется с помощью исходящих сервисов;
- подписка на события обрабатывается с помощью интерфейса и входящих сервисов.

Поскольку в настоящий момент мы находимся на стадии реализации модели предметной области (домена), единственным из перечисленных выше этапов, рассматриваемых в этом подразделе, является регистрация событий агрегатом. В следующих подразделах будут отдельно рассматриваться все прочие аспекты (исходящие сервисы отвечают за реализацию генерации и публикации событий, входящие сервисы реализуют подписку на эти события).

Регистрация событий

Для реализации этого этапа воспользуемся шаблонным классом `AbstractAggregateRoot`, предоставляемым платформой Spring Data. Этот шаблонный класс предоставляет возможность регистрировать возникающие события.

Рассмотрим пример реализации такого класса. В листинге 5.15 показан класс агрегата Cargo, который является расширением шаблонного класса `AbstractAggregateRoot`.

Листинг 5.15 ❖ Шаблонный класс AbstractAggregateRoot

```
package com.practicalddd.cargotracker.bookingms.domain.model.aggregates;
import javax.persistence.*;
import org.springframework.data.domain.AbstractAggregateRoot;
@Entity
public class Cargo extends AbstractAggregateRoot<Cargo> {
}
```

Следующий шаг – реализация регистрируемых событий агрегата, возникающих при изменении его состояния. Ранее уже отмечалось и демонстрировалось, что группа операций Command (команды), изменяющие состояние агрегата, является наиболее подходящим местом для регистрации событий данного агрегата. В агрегате Cargo две операции-команды: первая выполняется при заказе нового груза, вторая – при назначении маршрута заказанному грузу. Изменения состояния этого агрегата размещаются в обработчиках его команд: Cargo Booking (заказ груза) в конструкторе агрегата Cargo, Cargo Routing (назначение маршрута) в методе `assignToRoute()` агрегата Cargo. Реализация регистрации и генерации событий агрегата будет выполнена в этих двух методах с использованием метода `registerEvent()`, предоставляемого шаблонным классом `AbstractAggregateRoot`.

В листинге 5.16 показана реализация операции регистрации событий агрегата в методах обработчиков команд агрегата Cargo. Добавляется новый ме-

тод addDomainEvent(), который инкапсулирует метод registerEvent(). Метод принимает в качестве входного параметра обобщенный объект события (Generic Event Object), представляющий собой событие, которое должно быть зарегистрировано. В конструкторе агрегата и в методе assignToRoute() вызывается метод addDomainEvent() с указанием соответствующих событий, которые должны быть зарегистрированы, т. е. CargoBookedEvent и CargoRoutedEvent.

Листинг 5.16 ❖ Регистрация событий в корневом агрегате Cargo

```
package com.practicalddd.cargotracker.bookingms.domain.model.aggregates;
import javax.persistence.*;
import org.springframework.data.domain.AbstractAggregateRoot;

@Entity
public class Cargo extends AbstractAggregateRoot<Cargo> {
/**
 * Конструктор – используется для создания нового объекта заказанного груза Cargo.
 * Регистрирует событие заказа груза Cargo.
 * @param bookingId - Booking Identifier for the new Cargo
 * @param routeSpecification - Route Specification for the new Cargo
 */
  /**
   * Конструктор обработчик команды заказа нового груза. Устанавливает состояние
   * агрегата и регистрирует событие заказа груза CargoBooked.
   *
   */
  public Cargo( BookCargoCommand bookCargoCommand ) {
    this.bookingId = new BookingId( bookCargoCommand.getBookingId() );
    this.routeSpecification = new RouteSpecification(
                    new Location( bookCargoCommand.getOriginLocation() ),
                    new Location( bookCargoCommand.getDestLocation() ),
                    bookCargoCommand.getDestArrivalDeadline() );
    this.origin = routeSpecification.getOrigin();
    this.itinerary = CargoItinerary.EMPTY_ITINERARY; // Пустой план маршрута, так как
                                                     // этому грузу еще не был назначен маршрут.
    this.bookingAmount = bookingAmount;
    this.delivery = Delivery.derivedFrom( this.routeSpecification, this.itinerary,
                                          LastCargoHandledEvent.EMPTY );
    // Добавление события, которое должно быть сгенерировано при сохранении нового груза.
    addDomainEvent( new CargoBookedEvent(
                        new CargoBookedEventData( bookingId.getBookingId() ) ) );
  }

  /**
   * Назначение маршрута для нового груза.
   * Регистрирует событие назначения маршрута новому грузу CargoRouted.
   * @param itinerary
   */
    /**
     * Обработчик команды назначения маршрута Route Cargo. Устанавливает состояние
     * агрегата и регистрирует событие назначения маршрута CargoRouted.
     * @param routeCargoCommand
     */
    public void assignToRoute( RouteCargoCommand routeCargoCommand ) {
```

```
      this.itinerary = routeCargoCommand.getCargoItinerary();
      // Синхронная обработка с сохранением согласованности внутреннего состояния
      // агрегата Cargo.
      this.delivery = delivery.updateOnRouting( this.routeSpecification,
                                                this.itinerary );
      // Добавление события, которое должно быть сгенерировано при сохранении нового груза.
      addDomainEvent(new CargoRoutedEvent(
                        new CargoRoutedEventData( bookingId.getBookingId() ) ) );
    }

    /**
     * Метод регистрации заданного события.
     * @param event
     */
    public void addDomainEvent( Object event ) {
      registerEvent( event );
    }
}
```

В листинге 5.17 показана реализация класса `CargoBookedEvent`. Это обычный старый объект Java (POJO), в котором инкапсулированы данные события, т. е. `CargoBookedEventData`.

Листинг 5.17 ❖ Реализация класса события CargoBookedEvent

```
/**
 * Класс события заказа груза CargoBooked. Является оберткой для класса
CargoBookedEventData.
 */
public class CargoBookedEvent {
  CargoBookedEventData cargoBookedEventData;
  public CargoBookedEvent( CargoBookedEventData cargoBookedEventData ) {
    this.cargoBookedEventData = cargoBookedEventData;
  }

  public CargoBookedEventData getCargoBookedEventData() {
    return cargoBookedEventData;
  }
}
```

В листинге 5.18 показана реализация класса `CargoBookedEventData`. Это также обычный старый объект Java (POJO), содержащий данные события, в данном случае идентификатор заказа `BookingId`.

Листинг 5.18 ❖ Реализация класса CargoBookingEventData

```
/**
 * Данные события заказа груза CargoBooked.
 */
public class CargoBookedEventData {
  private String bookingId;
  public CargoBookedEventData( String bookingId ) {
    this.bookingId = bookingId;
  }
  public String getBookingId() { return this.bookingId; }
}
```

Для реализации классов `CargoRoutedEvent` и `CargoRoutedEventData` применяется точно такая же методика, как для рассмотренных выше аналогичных классов.

На рис. 5.25 показана диаграмма класса в рассматриваемой здесь реализации.

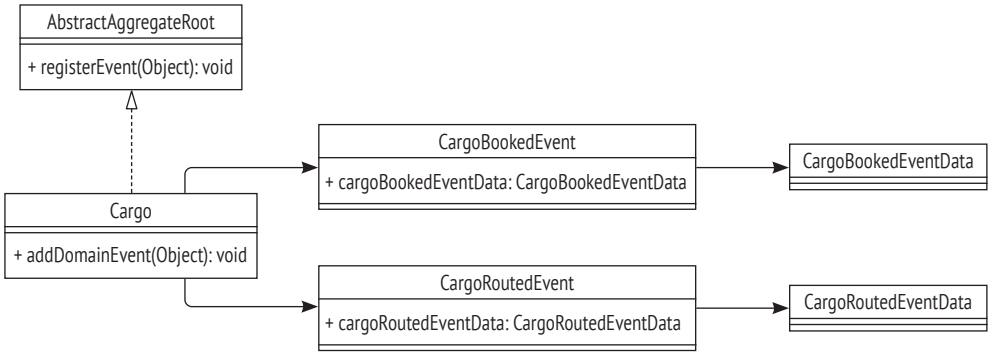

Рис. 5.25 ❖ Диаграмма класса, реализующего регистрацию события агрегата

Подводя итоги, отметим, что агрегаты регистрируют события предметной области (домена) после обработки команды. Регистрация этих событий всегда реализуется в методах обработчиков команд соответствующих агрегатов.

На этом завершается рассмотрение реализации модели предметной области (домена). Далее будут рассматриваться сервисы модели предметной области (домена).

Сервисы модели предметной области (домена)

Сервисы модели предметной области (домена) используются по двум основным причинам. Во-первых, это обеспечение доступности состояния ограниченного контекста для внешних потребителей через корректно определенные интерфейсы. Во-вторых, это обеспечение взаимодействия с внешними потребителями для сохранения состояния ограниченного контекста в хранилищах данных (базах данных), для публикации событий, соответствующих изменениям состояния ограниченного контекста во внешних брокерах сообщений, или для обмена данными с другими ограниченными контекстами.

Существует три типа сервисов модели предметной области (домена) для любого ограниченного контекста:

- входящие сервисы (Inbound Services), в которых реализуются корректно определенные интерфейсы, позволяющие внешним потребителям взаимодействовать с моделью предметной области (домена);
- исходящие сервисы (Outbound Services), в которых реализуются все взаимодействия с внешними репозиториями и другими ограниченными контекстами;
- сервисы приложения (Application Services), действующие как внешний уровень, связывающий модель предметной области (домена) с входящими и исходящими сервисами.

На рис. 5.26 показана реализация сервисов модели предметной области (домена).

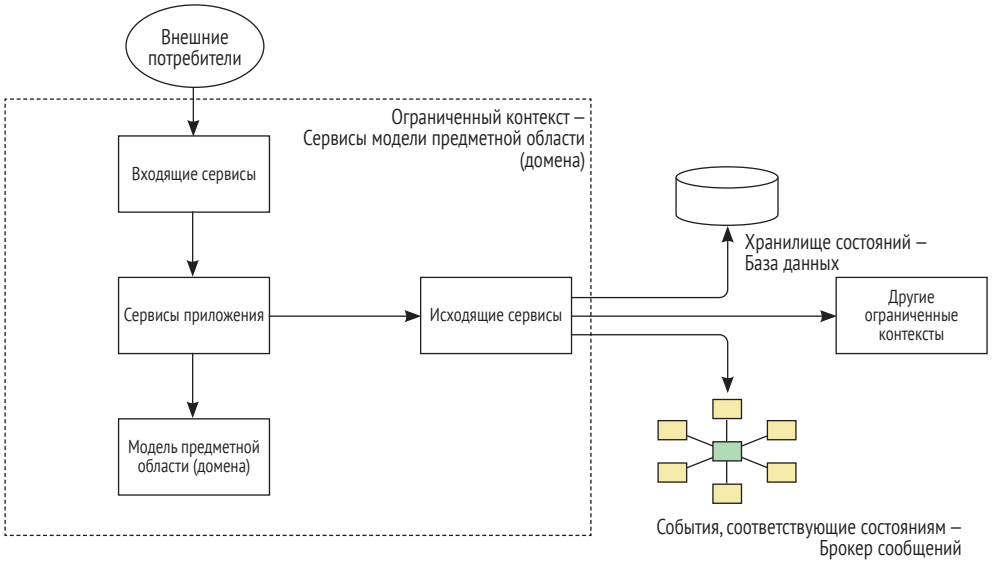

Рис. 5.26 ❖ Общая схема реализации сервисов модели предметной области (домена)

Входящие сервисы

Входящие сервисы (Inbound Services) (или входящие адаптеры (Inbound Adaptors) в соответствии с терминологией шаблона гексагональной архитектуры) действуют как внешний шлюз для модели основной предметной области (домена). Как уже было отмечено выше, здесь подразумевается реализация корректно определенных интерфейсов, позволяющих внешним потребителям взаимодействовать с моделью основной предметной области (домена).

Тип входящих сервисов зависит от типа операций, которые необходимо выполнить, чтобы разрешить взаимодействие внешних потребителей с моделью предметной области (домена).

С учетом реализации архитектурного шаблона микросервисов для приложения Cargo Tracker в рассматриваемом здесь примере определяются следующие два типа входящих сервисов:

- уровень API на основе REST, используемый внешними потребителями для вызова операций в ограниченном контексте (команды и запросы);
- уровень обработки событий на основе механизма Spring Cloud Stream, который извлекает события из брокера сообщений и обрабатывает их.

REST API

Обязанности уровня REST API как доверенного представителя ограниченного контекста состоят в приеме HTTP-запросов от внешних потребителей. HTTP-запрос может быть командой или запросом. Уровень REST API обязан преобра-

зовать внешний HTTP-запрос в модель команды или запроса, распознаваемую моделью предметной области (домена) ограниченного контекста, и передать ее на уровень сервисов приложения для дальнейшей обработки.

Еще раз обратимся к рис. 4.5, на котором подробно показаны все операции для различных ограниченных контекстов (например, заказ груза, назначение маршрута доставки груза, транспортная обработка груза, отслеживание груза и т. п.). Все эти операции будут иметь соответствующие интерфейсы REST API, которые принимают внешние запросы и обрабатывают их.

Реализация уровня REST API на платформе Spring Boot выполняется с использованием функциональных возможностей REST, предоставляемых проектом Spring Web MVC. Зависимость spring-boot-starter-web, добавляемая в проект, предоставляет все требуемые функциональные возможности для создания REST API.

Рассмотрим подробнее пример создания REST API с использованием платформы Spring Web. В листинге 5.19 показан класс `CargoBookingController`, предоставляющий интерфейс REST API для команды заказа груза (Cargo Booking Command). Ниже перечислены некоторые характеристики этого класса:

- интерфейс REST API доступен по URL */cargobooking*;
- класс содержит единственный метод POST, принимающий параметр `BookCargoResource`, который содержит входную полезную нагрузку для этого API;
- существует зависимость от сервиса приложения `CargoBookingCommandService`, который действует как внешний сервис (см. его реализацию ниже). Эта зависимость инъецирована в класс API с помощью механизма инъекции зависимостей на основе конструктора;
- класс выполняет преобразование данных ресурса `BookCargoResource` в модель команды `BookCargoCommand`, используя класс утилиты ассемблера `BookCargoCommandDTOAssembler`;
- после преобразования класс делегирует процесс в сервис `CargoBookingCommandService` для дальнейшей обработки;
- класс возвращает ответ внешнему потребителю с идентификатором заказа нового груза.

Листинг 5.19 ❖ Реализация класса CargoBookingController

```
package com.practicalddd.cargotracker.bookingms.interfaces.rest;

import com.practicalddd.cargotracker.bookingms.application.internal
          .commandservices.CargoBookingCommandService;
import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.BookingId;
import com.practicalddd.cargotracker.bookingms.interfaces.rest.dto.BookCargoResource;
import com.practicalddd.cargotracker.bookingms.interfaces.rest.transform
          .BookCargoCommandDTOAssembler;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

@Controller    // Признак того, что этот класс является контроллером.
@RequestMapping( "/cargobooking" ) // Указатель URI для этого интерфейса API.
public class CargoBookingController {
  private CargoBookingCommandService cargoBookingCommandService;
```

```
  // Зависимости от сервисов приложения.
  /**
   * Обеспечение зависимостей.
   * @param cargoBookingCommandService
   */
  public CargoBookingController( CargoBookingCommandService cargoBookingCommandService ) {
    this.cargoBookingCommandService = cargoBookingCommandService;
  }

  /**
   * POST-метод для заказа груза
   * @param bookCargoResource
   */
  @PostMapping
  @ResponseBody
  public BookingId bookCargo( @RequestBody  BookCargoResource bookCargoResource ) {
    BookingId bookingId  = cargoBookingCommandService.bookCargo(
                    BookCargoCommandDTOAssembler.toCommandFromDTO( bookCargoResource ) );
    return bookingId;
  }
}
```

В листинге 5.20 показана реализация класса `BookCargoResource`.

Листинг 5.20 ❖ Реализация класса BookCargoResource

```
package com.practicalddd.cargotracker.bookingms.interfaces.rest.dto;

import java.time.LocalDate;

/**
 * Класс ресурса для API команды заказа нового груза.
 */
public class BookCargoResource {
  private int bookingAmount;
  private String originLocation;
  private String destLocation;
  private LocalDate destArrivalDeadline;
  public BookCargoResource() {}
  public BookCargoResource( int bookingAmount, String originLocation, String destLocation,
                            LocalDate destArrivalDeadline ) {
    this.bookingAmount = bookingAmount;
    this.originLocation = originLocation;
    this.destLocation = destLocation;
    this.destArrivalDeadline = destArrivalDeadline;
  }

  public void setBookingAmount( int bookingAmount ) {
    this.bookingAmount = bookingAmount;
  }

  public int getBookingAmount() {
    return this.bookingAmount;
  }
```

```
  public String getOriginLocation() { return originLocation; }

  public void setOriginLocation( String originLocation ) {
    this.originLocation = originLocation;
  }

  public String getDestLocation() { return destLocation; }

  public void setDestLocation( String destLocation ) {
    this.destLocation = destLocation;
  }

  public LocalDate getDestArrivalDeadline() { return destArrivalDeadline; }

  public void setDestArrivalDeadline( LocalDate destArrivalDeadline ) {
    this.destArrivalDeadline = destArrivalDeadline;
  }
}
```

В листинге 5.21 показана реализация класса `BookCargoCommandDTOAssembler`.

Листинг 5.21 ❖ Реализация класса BookCargoCommandDTOAssembler

```
package com.practicalddd.cargotracker.bookingms.interfaces.rest.transform;

import com.practicalddd.cargotracker.bookingms.domain.model.commands.BookCargoCommand;
import com.practicalddd.cargotracker.bookingms.interfaces.rest.dto.BookCargoResource;

/**
 * Класс ассемблера для преобразования данных ресурса BookCargo в модель BookCargo.
 */
public class BookCargoCommandDTOAssembler {
  /**
   * Статический метод в классе ассемблера.
   * @param bookCargoResource
   * @return BookCargoCommand Model
   */
  public static BookCargoCommand toCommandFromDTO( BookCargoResource bookCargoResource ) {
    return new BookCargoCommand(
                  bookCargoResource.getBookingAmount(),
                  bookCargoResource.getOriginLocation(),
                  bookCargoResource.getDestLocation(),
                  java.sql.Date.valueOf(bookCargoResource.getDestArrivalDeadline() ) );
  }
}
```

В листинге 5.22 показана реализация класса `BookCargoCommand`.

Листинг 5.22 ❖ Реализация класса BookCargoCommand

```
package com.practicalddd.cargotracker.bookingms.domain.model.commands;

import java.util.Date;

/**
 * Класс команды заказа нового груза BookCargo.
 */
```

```
public class BookCargoCommand {
  private int bookingAmount;
  private String originLocation;
  private String destLocation;
  private Date destArrivalDeadline;
  public BookCargoCommand() {}
  public BookCargoCommand( int bookingAmount, String originLocation, String destLocation,
                           Date destArrivalDeadline) {
    this.bookingAmount = bookingAmount;
    this.originLocation = originLocation;
    this.destLocation = destLocation;
    this.destArrivalDeadline = destArrivalDeadline;
  }

  public void setBookingAmount( int bookingAmount ) {
    this.bookingAmount = bookingAmount;
  }

  public int getBookingAmount() {
    return this.bookingAmount;
  }

  public String getOriginLocation() { return originLocation; }

  public void setOriginLocation( String originLocation ) {
    this.originLocation = originLocation;
  }

  public String getDestLocation() { return destLocation; }

  public void setDestLocation( String destLocation ) {
    this.destLocation = destLocation;
  }

  public Date getDestArrivalDeadline() { return destArrivalDeadline; }

  public void setDestArrivalDeadline( Date destArrivalDeadline ) {
    this.destArrivalDeadline = destArrivalDeadline;
  }
}
```

На рис. 5.27 показана диаграмма класса для приведенной выше реализации.

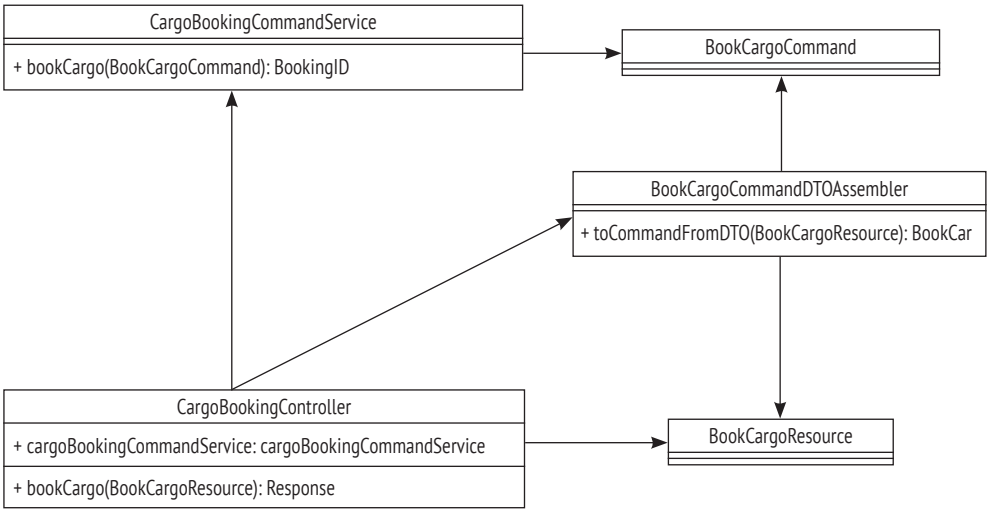

Рис. 5.27 ❖ Диаграмма класса для реализации REST API

Все реализации входящих REST API в рассматриваемом здесь примере используют одинаковую методику, которая показана на рис. 5.28.

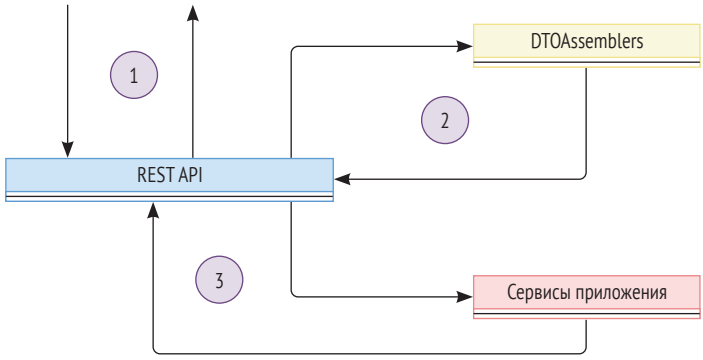

Рис. 5.28 ❖ Общая схема процесса реализации входящих сервисов

1. Входящий внешний запрос на выполнение команды или (внутреннего) запроса поступает в интерфейс REST API. Классы API реализуются с использованием проекта Spring Web MVC, который конфигурируется при добавлении зависимости *spring-boot-starter-web* в наш проект.
2. Класс REST API использует утилиту Assembler как компонент для преобразования формата данных ресурса в формат данных команды или запроса, требуемый моделью предметной области (домена).
3. Данные команды или запроса передаются в сервисы приложения для дальнейшей обработки.

Обработчики событий

Другим типом интерфейсов, существующих в ограниченных контекстах, являются обработчики событий (Event Handlers). В любом ограниченном контексте обработчики событий отвечают за обработку событий, которые в той или иной степени важны для данного конкретного ограниченного контекста. События генерируются другими ограниченными контекстами в том же приложении. Объекты типа `EventHandler` создаются в ограниченном контексте, который является подписчиком и размещается на уровне *inbound/interface*. Обработчики событий принимают событие (Event) вместе с данными полезной нагрузки этого события и обрабатывают их в форме обычной операции.

Реализация обработчиков событий выполняется с использованием функциональных возможностей, предоставляемых платформой Spring Cloud Stream. В качестве брокера сообщений выбран RabbitMQ, поэтому в рассматриваемой здесь реализации подразумевается, что экземпляр RabbitMQ уже инициализирован и работает. Нет никакой необходимости создавать какие-либо специализированные точки обмена, цели или очереди в брокере сообщений RabbitMQ.

Воспользуемся примером ограниченного контекста Tracking, который заинтересован в событии CargoRouted, публикуемом после обработки команды RouteCargo. Рассмотрим процесс реализации во всех подробностях:

1. Первый шаг – реализация класса обработчика как обычного класса сервиса со стандартной аннотацией `@Service`. Класс сервиса связывается с каналом соединения с брокером сообщений, используя аннотацию `@EnableBinding`. Наконец, маркируется метод обработчика события в классе обработчика, используя аннотацию `@StreamListener` с указанием подробной информации о цели. Эта аннотация помечает метод приема потока сообщений, публикуемых в целевом пункте, в котором заинтересован этот обработчик.

   В листинге 5.23 показана реализация класса обработчика `CargoRoutedEventHandler`.

Листинг 5.23 ❖ Реализация класса обработчика CargoRoutedEventHandler

```
package com.practicalddd.cargotracker.trackingms.interfaces.events;

import com.practicalddd.cargotracker.shareddomain.events.CargoRoutedEvent;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.cloud.stream.messaging.Sink;
import org.springframework.stereotype.Service;

/**
 * Обработчик события CargoRouted, в котором заинтересован ограниченный контекст Tracking.
 */
@Service
@EnableBinding( Sink.class ) // Связывание с каналом соединения с брокером сообщений.
```

```
public class CargoRoutedEventHandler {
  @StreamListener( target = Sink.INPUT ) // Прослушивание потока сообщений в целевом
пункте.
  public void receiveEvent( CargoRoutedEvent cargoRoutedEvent ) {
    // Обработка события.
  }
}
```

2. Также необходимо реализовать конфигурацию брокера сообщений в форме подробной информации о соединении с брокером и отображений целевых узлов брокера. В листинге 5.24 показана конфигурация, которую необходимо реализовать в файле *application.properties* приложения Spring Boot. В конфигурации свойств брокера содержатся значения по умолчанию, присваиваемые при первоначальной установке RabbitMQ.

Листинг 5.24 ❖ Конфигурация свойств брокера RabbitMQ

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
spring.cloud.stream.bindings.input.destination=cargoRoutings
spring.cloud.stream.bindings.input.group=cargoRoutingsQueue
```

Цель конфигурируется с тем же значением, которое используется при публикации события `CargoRouted` в ограниченном контексте Booking (см. раздел «Исходящие сервисы»).

На рис. 5.29 показана диаграмма класса для приведенной выше реализации.

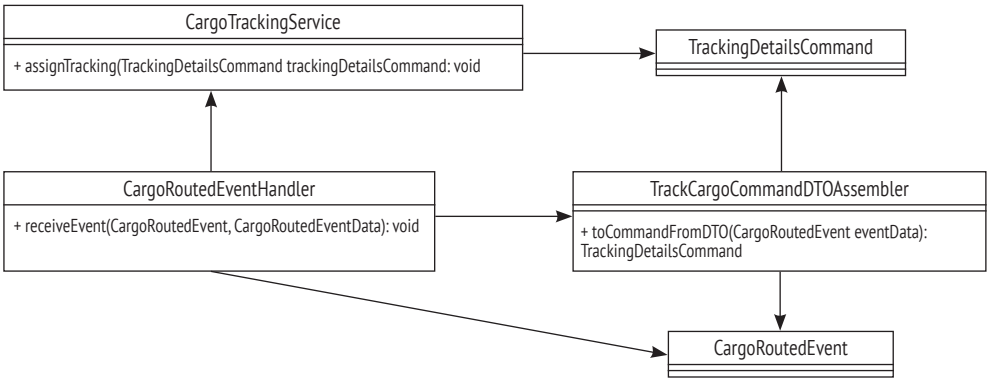

Рис. 5.29 ❖ Диаграмма класса для реализации обработчика событий

Все реализации обработчиков событий в рассматриваемом здесь примере используют одинаковую методику, которая показана на рис. 5.30.

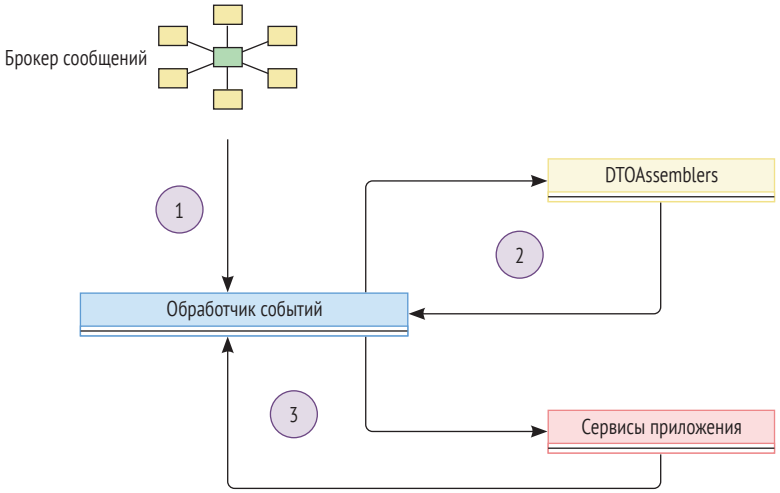

Рис. 5.30 ❖ Общая схема процесса реализации обработчиков событий

1. Обработчики событий принимают входящие события от брокера сообщений.
2. Обработчики событий используют утилиту Assembler как компонент для преобразования формата данных ресурса в формат данных команды, требуемый моделью предметной области (домена).
3. Данные команды передаются в сервисы приложения для дальнейшей обработки.

Сервисы приложения

Сервисы приложения действуют как внешний связующий уровень или порт между входящими/исходящими сервисами и моделью основной предметной области (домена) в ограниченном контексте.

В ограниченном контексте сервисы приложения отвечают за прием запросов от входящих сервисов и делегирование их соответствующим сервисам, т. е. команды передаются в сервисы команд, а запросы передаются сервисам запросов. Как часть процесса делегирования команд, сервисы приложения отвечают за постоянное сохранение состояния агрегата в базе данных более низкого уровня. Как часть процесса делегирования запросов, сервисы приложения отвечают за извлечения состояния агрегата из базы данных более низкого уровня.

При исполнении этих обязанностей сервисы приложения полагаются на исходящие сервисы для выполнения своих задач. Исходящие сервисы предоставляют необходимые компоненты инфраструктуры, требуемые для соединения с физическими хранилищами данных. Более подробно реализация исходящих сервисов будет рассматриваться отдельно (см. раздел «Исходящие сервисы»).

На рис. 5.31 показаны обязанности сервисов приложения.

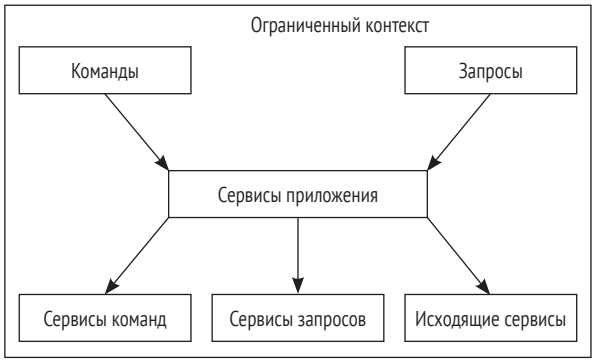

Рис. 5.31 ❖ Обязанности сервисов приложения

Сервисы приложения: делегирование команд и запросов

Как часть рассматриваемой здесь области ответственности, сервисы приложений в ограниченном контексте принимают внешние запросы (requests) на обработку команд и запросов. Обычно такие внешние запросы приходят из входящих сервисов (уровень API). Как часть этого процесса обработки, сервисы приложения сначала используют обработчики команд и запросов (см. соответствующий раздел по модели предметной области (домена)) модели предметной области (домена) для установки состояния или запроса о состоянии. Затем сервисы приложения используют исходящие сервисы для сохранения состояния или для выполнения запросов о состоянии агрегата.

Рассмотрим пример класса сервисов приложения, предназначенный для делегирования команд (Command Delegator), а именно класс сервисов приложения для команды заказа груза (Cargo Booking Command). Этот класс содержит два метода `bookCargo()` и `assignRouteToCargo()`, которые обрабатывают команды заказа груза и назначения маршрута заказанному грузу:

- класс сервисов приложения реализуется как обычный компонент Spring Managed Bean с маркером аннотации @Service, обозначающим класс сервиса;
- класс сервисов приложения обеспечивается всем необходимыми зависимостями с помощью функциональных возможностей механизма инъекции зависимостей в конструкторе. В рассматриваемом примере класс `CargoBookingCommandApplicationService` зависит от класса репозитория исходящего сервиса (`CargoRepository`);
- в обоих методах сервисы приложения полагаются на обработчики команд, определенные в агрегате Cargo (конструктор и `assignToRoute()`) для установки состояния этого агрегата;
- сервисы приложения используют исходящий сервис CargoRepository для сохранения состояния агрегата Cargo в обеих вышеназванных операциях.

В листинге 5.25 показана реализация класса сервиса приложения для обработки команды заказа нового груза.

Листинг 5.25 ❖ Реализация класса CargoBookingCommandApplicationService

```
package com.practicalddd.cargotracker.bookingms.application.internal.commandservices;

import   com.practicalddd.cargotracker.bookingms.application.internal.outboundservices
           .acl.ExternalCargoRoutingService;
import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.BookingId;
import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.Cargo;
import com.practicalddd.cargotracker.bookingms.domain.model.commands.BookCargoCommand;
import com.practicalddd.cargotracker.bookingms.domain.model.commands.RouteCargoCommand;
import com.practicalddd.cargotracker.bookingms.domain.model.entities.Location;
import com.practicalddd.cargotracker.bookingms.domain.model.valueobjects.CargoItinerary;
import com.practicalddd.cargotracker.bookingms.domain.model.valueobjects.
RouteSpecification;
import com.practicalddd.cargotracker.bookingms.infrastructure.repositories.CargoRepository;
import org.springframework.stereotype.Service;

import java.util.UUID;

/**
 * Класс сервиса приложения для команды заказа нового груза.
 */
@Service
public class CargoBookingCommandService {
  private CargoRepository cargoRepository;
  private ExternalCargoRoutingService externalCargoRoutingService;
  public CargoBookingCommandService( CargoRepository cargoRepository ) {
    this.cargoRepository = cargoRepository;
    this.externalCargoRoutingService = externalCargoRoutingService;
  }

  /**
   * Метод сервиса команд для заказа нового груза.
   * @return BookingId of the Cargo
   */
  public BookingId bookCargo( BookCargoCommand bookCargoCommand ) {
    String random = UUID.randomUUID().toString().toUpperCase();
    bookCargoCommand.setBookingId( random );
    Cargo cargo = new Cargo( bookCargoCommand );
    cargoRepository.save( cargo );
    return new BookingId( random );
  }

  /**
   * Метод сервиса команд для назначения маршрута для нового груза.
   * @param routeCargoCommand
   */
  public void assignRouteToCargo( RouteCargoCommand routeCargoCommand ) {
    Cargo cargo = cargoRepository.findByBookingId( routeCargoCommand.getCargoBookingId() );
    CargoItinerary cargoItinerary =
        externalCargoRoutingService.fetchRouteForSpecification( new RouteSpecification(
                new Location( routeCargoCommand.getOriginLocation() ),
                new Location( routeCargoCommand.getDestinationLocation() ),
                routeCargoCommand.getArrivalDeadline() ) );
```

```
    routeCargoCommand.setCargoItinerary( cargoItinerary );
    cargo.assignToRoute( routeCargoCommand );
    cargoRepository.save( cargo );
  }
}
```

В листинге 5.26 показана реализация класса сервисов приложения `CargoBookingQuery`, который обслуживает все запросы, относящиеся к ограниченному контексту Booking.

Листинг 5.26 ❖ Реализация сервисов приложения CargoBookingQuery

```
package com.practicalddd.cargotracker.bookingms.application.internal.queryservices;

import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.BookingId;
import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.Cargo;
import com.practicalddd.cargotracker.bookingms.infrastructure.repositories.CargoRepository;
import org.springframework.stereotype.Service;

import java.util.List;

/**
 * Сервис приложения, обслуживающий все запросы, относящиеся к ограниченному
 * контексту Booking.
 */
@Service
public class CargoBookingQueryService {
  private CargoRepository cargoRepository; // Инъекция зависимостей.

    /**
     * Поиск всех новых заказанных грузов.
     * @return List<Cargo>
     */
    public List<Cargo> findAll() {
      return cargoRepository.findAll();
    }

    /**
     * Список всех идентификаторов заказанных грузов.
     * @return List<BookingId>
     */
   public List<BookingId> getAllBookingIds() {
     return cargoRepository.findAllBookingIds();
   }

    /**
     * Поиск конкретного груза по его идентификатору BookingId.
     * @param bookingId
     * @return Cargo
     */
    public Cargo find( String bookingId ) {
      return cargoRepository.findByBookingId( bookingId );
    }
}
```

На рис. 5.32 показана диаграмма класса для приведенной выше реализации.

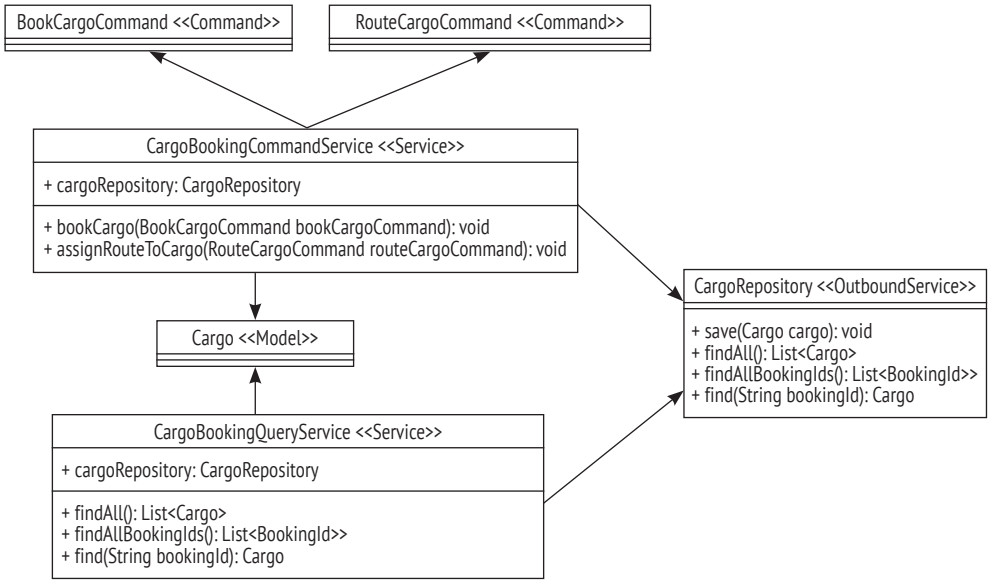

Рис. 5.32 ❖ Диаграмма класса сервисов приложения, делегирующего команды и запросы

При реализации всех сервисов приложения, отвечающих за делегирование команд и запросов, в рассматриваемом здесь примере используется один и тот же подход, показанный на рис. 5.33.

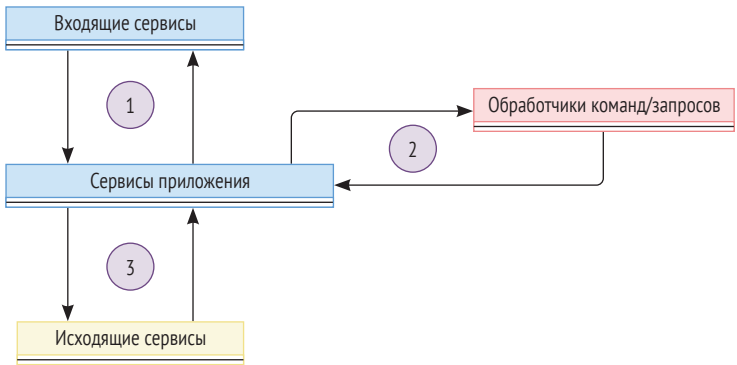

Рис. 5.33 ❖ Общая схема реализации сервисов приложения

1. Внешний запрос (request) на выполнение команды/запроса поступает в сервисы приложения ограниченного контекста. Этот внешний запрос обычно передается из входящих сервисов. Классы сервисов приложения реализуются как компоненты Spring Managed Beans с помощью маркера аннотации `@Service`. Классы сервисов приложения содержат все необходимые зависимости, инъецируемые через конструктор.
2. Сервисы приложения полагаются на обработчики команд и запросов, определенные в модели предметной области (домена), для установления и запроса состояния агрегата.
3. Сервисы приложения используют исходящие сервисы (например, репозитории) для обеспечения персистентности состояния агрегата или для выполнения запроса к агрегату.

Исходящие сервисы

Рассматривая реализацию сервисов приложения в предыдущем разделе, мы обратили внимание на то, что при обработке команд и запросов сервисам приложения может потребоваться обмен данными со следующими внешними сервисами:

- репозиториями – для сохранения или извлечения состояния ограниченного контекста;
- брокерами сообщений – для передачи информации об изменениях состояния ограниченного контекста;
- другими ограниченными контекстами.

Сервисы приложения связаны с исходящими сервисами для обеспечения перечисленных выше вариантов обмена данными.

Исходящие сервисы предоставляют функциональные возможности для взаимодействия с сервисами, являющимися внешними по отношению к ограниченному контексту. Таким внешним сервисом может быть хранилище данных, где содержится состояние агрегата ограниченного контекста или брокер сообщений, в котором публикуется состояние агрегата. К внешним сервисам также относится взаимодействие с другим ограниченным контекстом.

На рис. 5.34 показаны обязанности исходящих сервисов. Исходящие сервисы принимают внешние запросы (requests) на обмен данными с внешними сервисами как части операции (команды, запросы, события). Исходящие сервисы используют API (API персистентности, REST API, API брокера) в соответствии с типом внешнего сервиса, с которым необходимо взаимодействие.

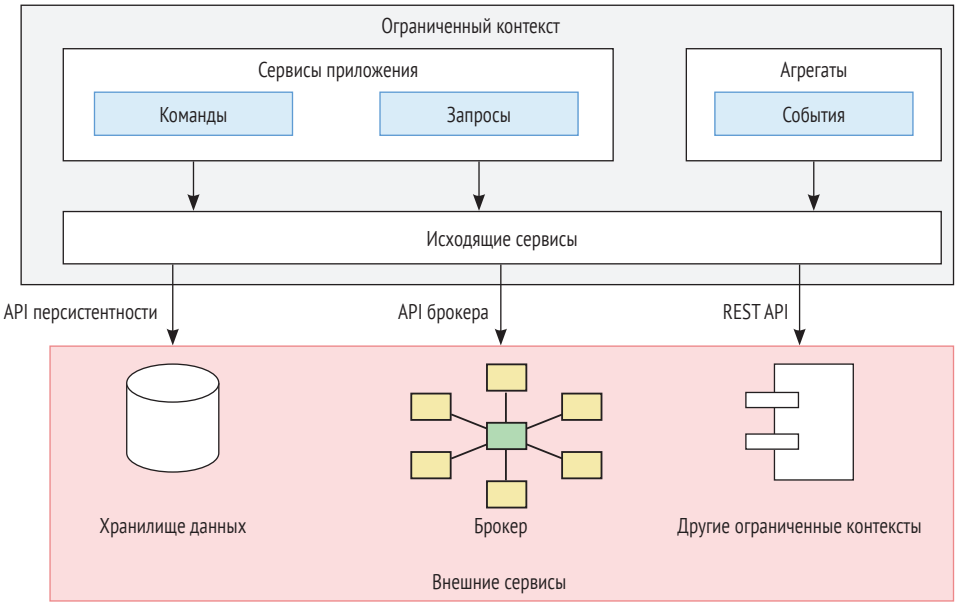

Рис. 5.34 ❖ Исходящие сервисы

Рассмотрим подробнее реализации различных типов исходящих сервисов.

Исходящие сервисы: классы репозиториев

Исходящие сервисы для доступа к базе данных реализуются как классы `Repository`. Класс репозитория создается для конкретного агрегата и выполняет все операции базы данных для этого агрегата, в том числе следующие:

- обеспечение персистентности нового агрегата и всех связанных с ним объектов;
- обновление агрегата и всех связанных с ним объектов;
- обслуживание запросов к агрегату и к связанным с ним объектам.

Платформа Spring Data JPA упрощает реализацию классов репозиториев JPA. Рассмотрим пример класса репозитория `CargoRepository`, который выполняет все операции базы данных, связанные с агрегатом Cargo:

- класс репозитория реализуется как интерфейс, являющийся расширением интерфейса `JpaRepository<T,ID>`;
- платформа Spring Data JPA обеспечивает автоматическую реализацию базовых операций CRUD (создание, чтение, обновление, удаление), необходимых для агрегата Cargo;
- добавляются только методы, требуемые для любого типа специализированных запросов, которые отображаются в соответствующие именованные запросы, определенные в агрегате Cargo.

В листинге 5.27 показана реализация класса `CargoRepository`.

Листинг 5.27 ❖ Реализация интерфейса JPA CargoRepository

```
package com.practicalddd.cargotracker.bookingms.infrastructure.repositories;

import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.BookingId;
```

```
import com.practicalddd.cargotracker.bookingms.domain.model.aggregates.Cargo;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

/**
 * Класс репозитория для агрегата Cargo.
 */
public interface CargoRepository extends JpaRepository<Cargo, Long> {
  Cargo findByBookingId( String BookingId );
  List<BookingId> findAllBookingIds();
  List<Cargo> findAll();
}
```

На рис. 5.35 показана диаграмма класса для приведенной выше реализации.

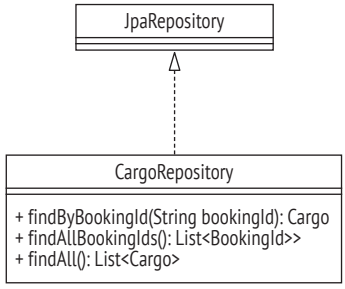

Рис. 5.35 ❖ Реализация исходящих сервисов – класс репозитория

По этой методике реализуются все классы репозиториев в рассматриваемом здесь примере.

Исходящие сервисы: REST API

Использование REST API как режима обмена данными между микросервисами – это достаточно часто встречающееся общее техническое требование. Ранее мы рассматривали хореографию событий как механизм для достижения этой цели, но иногда синхронный вызов между ограниченными контекстами также может являться техническим требованием.

Рассмотрим этот подход на конкретном примере. Как часть процесса заказа нового груза необходимо сформировать для этого груза план маршрута доставки, зависящий от спецификации назначенного маршрута. Данные, требуемые для генерации оптимального плана маршрута доставки, обрабатываются как часть ограниченного контекста Routing (назначение маршрута), в котором выполняется обработка перемещений, планов и графиков для транспортных средств (судов). При этом требуется, чтобы сервис заказа груза ограниченного контекста Booking выполнил исходящий вызов сервиса маршрутизации ограниченного контекста Routing, который предоставляет REST API для извлечения всех возможных планов маршрута доставки, зависимых от спецификации маршрута этого груза.

Этот процесс показан на рис. 5.36.

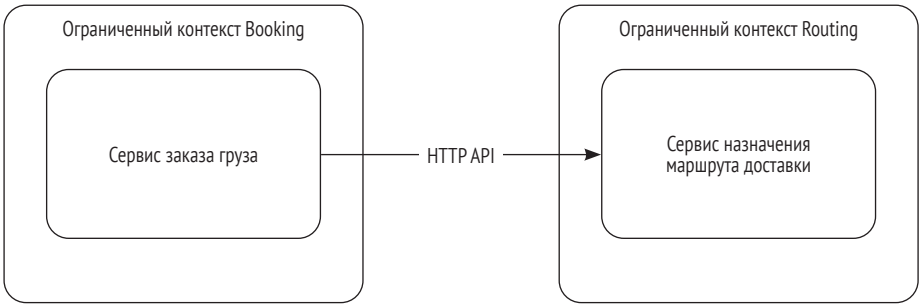

Рис. 5.36 ❖ HTTP-вызов между двумя ограниченными контекстами

Но при этом возникает затруднение с точки зрения модели предметной области (домена). Агрегат Cargo ограниченного контекста Booking содержит представление плана маршрута доставки как объект `CargoItinerary`, в то время как в ограниченном контексте Routing план маршрута доставки представлен как объект `TransitPath`. Таким образом, вызов между этими двумя ограниченными контекстами потребует преобразования типов объектов между соответствующими моделями предметных областей (доменов).

Такое преобразование обычно выполняется на уровне Anti-corruption (защита от повреждений), который действует как упрощенный шлюз (bridge) для обмена данными между двумя ограниченными контекстами, как показано на рис. 5.37.

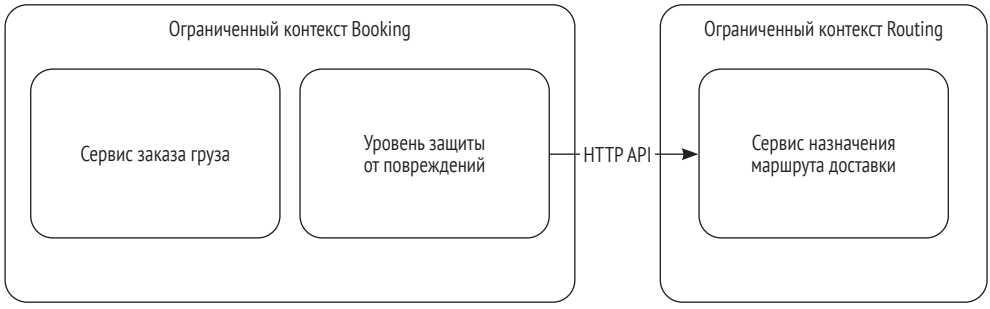

Рис. 5.37 ❖ Уровень защиты от повреждений между двумя ограниченными контекстами

Ограниченный контекст Booking полагается на функциональные возможности шаблона Rest, предоставляемого платформой Spring Web для вызова REST API сервиса назначения маршрута доставки.

Чтобы лучше понять изложенную концепцию, рассмотрим полную ее реализацию.

Первый шаг – реализация REST API сервиса назначения маршрута доставки груза. Для этого используются стандартные функциональные возможности Spring Web, которые уже были реализованы ранее в этой главе. В листинге 5.28 показана реализация REST API сервиса назначения маршрута доставки груза:

Листинг 5.28 ❖ Реализация класса контроллера CargoRoutingController

```
package com.practicalddd.cargotracker.routingms.interfaces.rest;

import com.practicalddd.cargotracker.TransitPath;
import com.practicalddd.cargotracker.routingms.application.internal.CargoRoutingService;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

@Controller    // Эта аннотация определяет, что данный класс является контроллером.
@RequestMapping( "/cargorouting" )
public class CargoRoutingController {
  private CargoRoutingService cargoRoutingService;
  // Зависимость от сервисов приложения.
    /**
     * Обеспечение зависимостей.
     * @param cargoRoutingService
     */
    public CargoRoutingController( CargoRoutingService cargoRoutingService ) {
      this.cargoRoutingService = cargoRoutingService;
    }

    /**
     *
     * @param originUnLocode
     * @param destinationUnLocode
     * @param deadline
     * @return TransitPath - Оптимальный маршрут для заданной спецификации.
     */
    @GetMapping( path = "/optimalRoute" )
    @ResponseBody
    public TransitPath findOptimalRoute(
             @PathVariable( "origin" ) String originUnLocode,
             @PathVariable( "destination" ) String destinationUnLocode,
             @PathVariable( "deadline" ) String deadline) {

       TransitPath transitPath = cargoRoutingService.findOptimalRoute( originUnLocode,
       destinationUnLocode, deadline );

       return transitPath;
    }
}
```

Реализация сервиса назначения маршрута доставки предоставляет интерфейс REST API, доступный в локации */optimalRoute*. Этот интерфейс принимает набор данных спецификации – исходную локацию, пункт назначения, предельный срок доставки. Затем используется класс сервисов приложения `CargoRouting` для вычисления оптимального плана маршрута на основе принятых данных спецификации. Модель предметной области (домена) в ограниченном контексте Routing представляет оптимальный план маршрута доставки в терминах `TransitPath` (это аналог плана маршрута – Itinerary) и `TransitEdge` (это аналог этапа – Leg).

В листинге 5.29 показана реализация класса `TransitPath` модели предметной области (домена).

Листинг 5.29 ❖ Реализация класса модели предметной области TransitPath

```
import java.util.ArrayList;
import java.util.List;

/**
 * Представление модели предметной области TransitPath.
 */
public class TransitPath {
  private List<TransitEdge> transitEdges;

  public TransitPath() {
    this.transitEdges = new ArrayList<>();
  }

  public TransitPath( List<TransitEdge> transitEdges ) {
    this.transitEdges = transitEdges;
  }

  public List<TransitEdge> getTransitEdges() {
    return transitEdges;
  }

  public void setTransitEdges( List<TransitEdge> transitEdges ) {
    this.transitEdges = transitEdges;
  }

  @Override
  public String toString() {
    return "TransitPath{" + "transitEdges=" + transitEdges + '}';
  }
}
```

В листинге 5.30 показана реализация класса `TransitEdge` модели предметной области (домена).

Листинг 5.30 ❖ Реализация класса TransitEdge модели предметной области (домена)

```
package com.practicalddd.cargotracker;

import java.io.Serializable;
import java.util.Date;

/**
 * Представляет ребро (этап) пути в графе, описывающем маршрут доставки
 * груза.
 */
public class TransitEdge implements Serializable {
  private String voyageNumber;
  private String fromUnLocode;
  private String toUnLocode;
  private Date fromDate;
  private Date toDate;

  public TransitEdge() {}

  public TransitEdge( String voyageNumber, String fromUnLocode, String toUnLocode,
```

```
                    Date fromDate, Date toDate ) {
    this.voyageNumber = voyageNumber;
    this.fromUnLocode = fromUnLocode;
    this.toUnLocode = toUnLocode;
    this.fromDate = fromDate;
    this.toDate = toDate;
  }

  public String getVoyageNumber() {
    return voyageNumber;
  }

  public void setVoyageNumber( String voyageNumber ) {
    this.voyageNumber = voyageNumber;
  }

  public String getFromUnLocode() {
    return fromUnLocode;
  }

  public void setFromUnLocode( String fromUnLocode ) {
    this.fromUnLocode = fromUnLocode;
  }

  public String getToUnLocode() {
    return toUnLocode;
  }

  public void setToUnLocode( String toUnLocode ) {
    this.toUnLocode = toUnLocode;
  }

  public Date getFromDate() {
    return fromDate;
  }

  public void setFromDate( Date fromDate ) {
    this.fromDate = fromDate;
  }

  public Date getToDate() {
    return toDate;
  }

  public void setToDate( Date toDate ) {
    this.toDate = toDate;
  }

  @Override
  public String toString() {
    return "TransitEdge{" + "voyageNumber=" + voyageNumber
                          + ", fromUnLocode=" + fromUnLocode + ", toUnLocode="
                          + toUnLocode + ", fromDate=" + fromDate
                          + ", toDate=" + toDate + '}';
  }
}
```

На рис. 5.38 показана диаграмма класса для приведенной выше реализации.

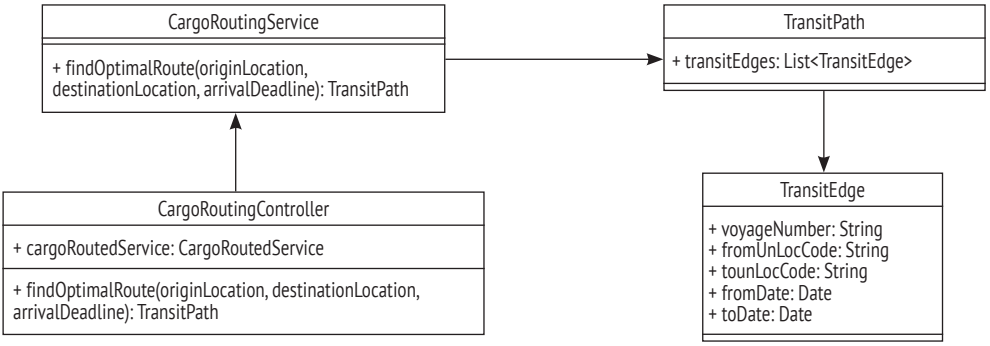

Рис. 5.38 ❖ Диаграмма класса для REST API

Следующий шаг – реализация на стороне клиента REST-сервиса для ограниченного контекста Routing (назначение маршрута доставки груза). Клиентом является класс `CargoBookingCommandService`, который отвечает за обработку команды назначения маршрута доставки груза (Assign Route to Cargo). Как части процесса обработки этой команды, этому классу сервиса потребуется вызов REST API сервиса назначения маршрута доставки для получения оптимального маршрута на основе спецификации маршрута для заказанного груза.

Класс `CargoBookingCommandService` использует класс исходящего сервиса `ExternalCargoRoutingService` для вызова REST API сервиса назначения маршрута доставки. Класс `ExternalCargoRoutingService` также выполняет преобразование данных, передаваемых из REST API сервиса назначения маршрута доставки, в формат, распознаваемый моделью предметной области (домена) ограниченного контекста Booking (заказ груза).

В листинге 5.31 показана реализация метода `assignRouteToCargo` в классе `CargoBookingCommandService`. Этот класс сервиса команд инъецируется с зависимостью от класса `ExternalCargoRoutingService`, который обрабатывает внешний запрос (request) для вызова REST API сервиса назначения маршрута доставки и возвращает объект `CargoItinerary`, т. е. план маршрута доставки, назначаемый заказанному новому грузу.

Листинг 5.31 ❖ Обеспечение зависимости от класса исходящего сервиса

```
@ApplicationScoped
public class CargoBookingCommandService {
  @Inject
  private ExternalCargoRoutingService externalCargoRoutingService;

  /**
   * Метод сервиса команд для назначения маршрута доставки груза.
   * @param routeCargoCommand
   */
  @Transactional
  public void assignRouteToCargo( RouteCargoCommand routeCargoCommand ) {
    Cargo cargo =
```

```
        cargoRepository.find( new BookingId(routeCargoCommand.getCargoBookingId() ) );
    CargoItinerary cargoItinerary =
        externalCargoRoutingService.fetchRouteForSpecification( new RouteSpecification(
                        new Location( routeCargoCommand.getOriginLocation() ),
                        new Location( routeCargoCommand.getDestinationLocation() ),
                        routeCargoCommand.getArrivalDeadline() ) );

    cargo.assignToRoute( cargoItinerary );
    cargoRepository.store( cargo );
  }

  // Реализация всех прочих команд для ограниченного контекста Booking.

}
```

В листинге 5.32 показан класс исходящего сервиса `ExternalCargoRoutingService`. Этот класс выполняет две операции:

- использует класс RestTemplate, предоставляемый проектом Spring Web. Этот класс помогает создавать REST-клиенты;
- выполняет преобразование данных, передаваемых REST API сервиса назначения маршрута доставки (`TransitPath`, `TransitEdge`) в формат, распознаваемый моделью предметной области (домена) ограниченного контекста Booking (`CargoItinerary`, `Leg`).

Листинг 5.32 ❖ Реализация класса исходящего сервиса

```
package com.practicalddd.cargotracker.bookingms.application.internal.outboundservices.acl;

import com.practicalddd.cargotracker.bookingms.domain.model.valueobjects.CargoItinerary;
import com.practicalddd.cargotracker.bookingms.domain.model.valueobjects.Leg;
import com.practicalddd.cargotracker.bookingms.domain.model.valueobjects.
RouteSpecification;
import com.practicalddd.cargotracker.shareddomain.TransitEdge;
import com.practicalddd.cargotracker.shareddomain.TransitPath;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
* Класс сервиса защиты от повреждений.
 */
@Service
public class ExternalCargoRoutingService {
  /**
   * Ограниченный контекст Booking выполняет внешний вызов сервиса назначения маршрута
   * доставки из ограниченного контекста Routing для получения и назначения оптимального
   * плана маршрута доставки заказанного груза на основе спецификации маршрута.
   * @param routeSpecification
   * @return
   */
  public CargoItinerary fetchRouteForSpecification( RouteSpecification
```

```
                                                    routeSpecification ) {
    RestTemplate restTemplate = new RestTemplate();
    Map<String,Object> params = new HashMap<>();
    params.put( "origin",routeSpecification.getOrigin().getUnLocCode() );
    params.put( "destination",routeSpecification.getDestination().getUnLocCode() );
    params.put( "arrivalDeadline",routeSpecification.getArrivalDeadline().toString() );
    TransitPath transitPath = restTemplate.getForObject( "<<ROUTING_SERVICE_URL>>/
                                                          cargorouting/",
                                                          TransitPath.class,params );
    List<Leg> legs = new ArrayList<>( transitPath.getTransitEdges().size() );

    for( TransitEdge edge : transitPath.getTransitEdges() ) {
      legs.add( toLeg( edge ) );
    }

    return new CargoItinerary(legs);
  }

  /**
   * Уровень защиты от повреждений: метод преобразования данных из формата модели домена
   * Routing (TransitEdges) в формат модели домена, распознаваемый ограниченным
   * контекстом Booking (Legs).
   * @param edge
   * @return
   */
  private Leg toLeg( TransitEdge edge ) {
    return new Leg( edge.getVoyageNumber(),
                    edge.getFromUnLocode(),
                    edge.getToUnLocode(),
                    edge.getFromDate(),
                    edge.getToDate() );
  }
}
```

На рис. 5.39 показана диаграмма класса для приведенной здесь реализации.

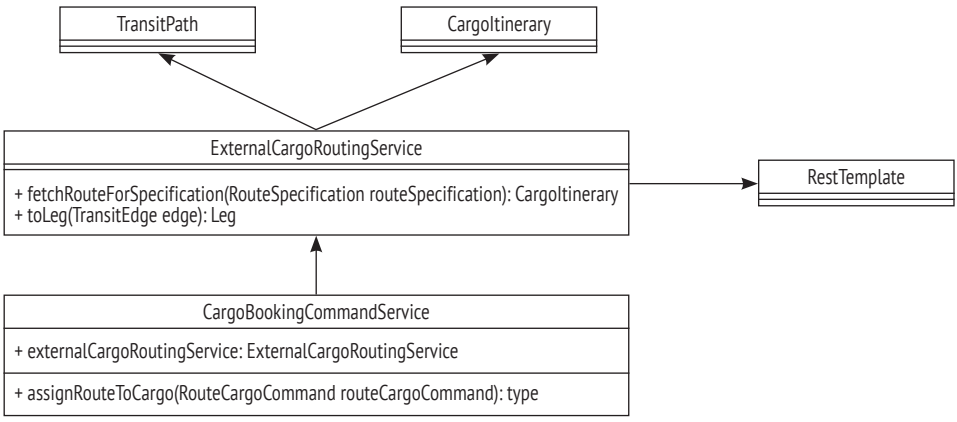

Рис. 5.39 ❖ Диаграмма класса
для реализации исходящих сервисов – REST API

Все реализации исходящих сервисов, для которых требуется обмен данными с другими ограниченными контекстами, следуют одной и той же методике, показанной на рис. 5.40.

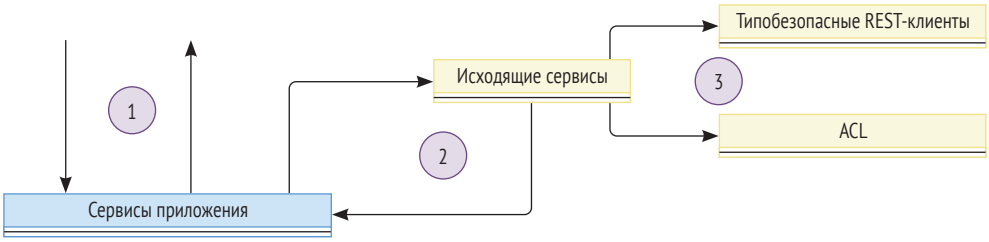

Рис. 5.40 ❖ Общая схема реализации исходящих сервисов (HTTP)

1. Классы сервисов приложения принимают команды/запросы/события.
2. Как часть процесса обработки, если требуется взаимодействие с API другого ограниченного контекста с применением REST, используется исходящий сервис.
3. Исходящий сервис использует класс RestTemplate для создания REST-клиента для вызова API требуемого ограниченного контекста. Исходящий сервис также выполняет преобразование данных из формата, предоставляемого API другого ограниченного контекста, в формат (модель) данных, распознаваемый текущим ограниченным контекстом.

Исходящие сервисы: брокер сообщений

Последним типом исходящих сервисов, который необходимо реализовать, является механизм генерации и публикации событий предметной области (домена), регистрируемых агрегатом во время обработки команд.

На рис. 5.41 показана полная схема механизма обработки потока событий в любом ограниченном контексте.

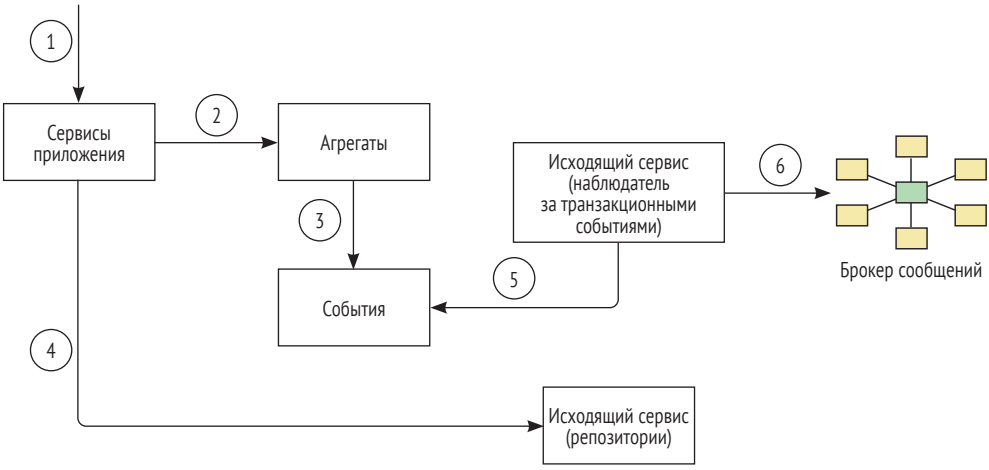

Рис. 5.41 ❖ Механизм обработки потока событий в ограниченном контексте

Рассмотрим подробнее последовательность событий.

1. Сервисы приложения принимают внешние запросы (requests) для обработки конкретной команды (например, заказ груза, назначение маршрута).
2. Сервисы приложения делегируют обработку обработчикам команд агрегата.
3. Обработчики команд регистрируют событие (например, груз заказан, маршрут назначен), которое необходимо опубликовать.
4. Сервисы приложения обеспечивают персистентность состояния агрегата, используя для этого репозитории исходящих сервисов.
5. Операция с репозиторием активизирует наблюдателей событий (Event Listeners) в исходящих сервисах. Наблюдатели событий собирают все ожидаемые зарегистрированные события предметной области (домена), которые должны быть опубликованы.
6. Наблюдатели событий публикуют события предметной области (домена) во внешнем брокере сообщений (т. е. RabbitMQ) в ходе той же самой транзакции.

Реализация наблюдателей событий выполняется с использованием функциональных возможностей, предоставляемых платформой Spring Cloud Stream. В качестве брокера сообщений выбран RabbitMQ, поэтому в рассматриваемой здесь реализации предполагается, что экземпляр RabbitMQ уже установлен, настроен и нормально работает. Нет необходимости создавать какие-либо специализированные точки обмена, цели или очереди в брокере сообщений RabbitMQ.

Продолжим рассмотрение примера ограниченного контекста Booking, где необходимо опубликовать события Cargo Booked Event и Cargo Routed Event после завершения обработки команд заказа груза (Book Cargo Command) и назначения маршрута доставки (Route Cargo Command).

Первый шаг – реализация источника этого события. Источник события содержит подробную информацию об исходящих каналах (логических соединениях) для рассматриваемых здесь событий.

В листинге 5.33 показана реализация источника события `CargoEventSource`. Здесь созданы два канала исходящих сообщений (cargoBookingChannel, cargoRoutingChannel).

Листинг 5.33 ❖ Реализация класса источника события

```
package com.practicalddd.cargotracker.bookingms.infrastructure.brokers.rabbitmq;

import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;

/**
 * Интерфейс, определяющий все исходящие каналы.
 */
public interface CargoEventSource {
  @Output( "cargoBookingChannel" )
  MessageChannel cargoBooking();
  @Output( "cargoRoutingChannel" )
  MessageChannel cargoRouting();
}
```

Второй шаг – реализация наблюдателя события (event listener). Источник события содержит подробную информацию о каналах вывода (логических соединениях – logical connections) для рассматриваемого в этом примере события.

В листинге 5.34 показана реализация класса CargoEventPublisherService. Это наблюдатель (слушатель – listener) всех событий предметной области (домена), зарегистрированных агрегатом Cargo и публикуемых им в брокере сообщений.

Реализация наблюдателя событий включает следующие шаги:

1) наблюдатель событий реализуется как обычный компонент Spring Managed Bean со стандартной аннотацией `@Service`. Эта реализация показана в листинге 5.34;
2) наблюдатель событий связывается с источником события, созданным на первом шаге, и использует для этого аннотацию `@EnableBinding`;
3) для каждого типа события предметной области (домена), зарегистрированного агрегатом Cargo, имеется соответствующая подпрограмма обработки в наблюдателе: например, для события `CargoBookedEvent` предусмотрен метод `handleCargoBooked()`, а для события `CargoRoutedEvent` – метод `handleCargoRouted()`. Эти подпрограммы (методы) в качестве входного параметра принимают соответствующее зарегистрированное событие;
4) эти подпрограммы (методы) маркируются аннотацией `@TransactionalEventListener`, обозначающей, что метод должен быть частью той же транзакции, что и операция с репозиторием;
5) в этих подпрограммах (методах) зарегистрированное событие публикуется в соответствующем канале брокера сообщений.

Листинг 5.34 ❖ Реализация класса наблюдателя событий

```
package com.practicalddd.cargotracker.bookingms.application.internal.outboundservices;

import com.practicalddd.cargotracker.bookingms.infrastructure.brokers.rabbitmq.
CargoEventSource;
import com.practicalddd.cargotracker.shareddomain.events.CargoBookedEvent;
import com.practicalddd.cargotracker.shareddomain.events.CargoRoutedEvent;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.event.TransactionalEventListener;

/**
 * Транзакционный наблюдатель событий для всех событий агрегата Cargo.
 */
@Service
@EnableBinding( CargoEventSource.class ) // Связывание с источником события.
public class CargoEventPublisherService {
  CargoEventSource cargoEventSource;
  public CargoEventPublisherService( CargoEventSource cargoEventSource ) {
    this.cargoEventSource = cargoEventSource;
  }

  @TransactionalEventListener // Связывание с транзакцией операции с репозиторием.
  public void handleCargoBookedEvent( CargoBookedEvent cargoBookedEvent ) {
      cargoEventSource.cargoBooking().send( MessageBuilder.
      withPayload(cargoBookedEvent).build() ); // Публикация события.
```

```
  }

  @TransactionalEventListener
  public void handleCargoRoutedEvent( CargoRoutedEvent cargoRoutedEvent ) {
    cargoEventSource.cargoRouting().send( MessageBuilder.
    withPayload(cargoRoutedEvent).build() );
  }
}
```

В дополнение к коду реализации также необходим код конфигурации брокера сообщений, как, например, подробная информация о соединении с брокером и об отображениях каналов в точки обмена в брокере. В листинге 5.35 показана конфигурация, которую необходимо реализовать в файле *application.properties* приложения Spring Boot. Параметры свойств для конфигурации брокера имеют значения по умолчанию, присваиваемые при первоначальной установке брокера сообщений RabbitMQ.

Листинг 5.35 ❖ Подробности конфигурации брокера сообщений RabbitMQ

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
spring.cloud.stream.bindings.cargoBookingChannel.destination=cargoBookings
spring.cloud.stream.bindings.cargoRoutingChannel.destination=cargoRoutings
```

Все исходящие сервисы, которые должны публиковать события предметной области (домена), следуют одной и той же методике, описанной выше.

На рис. 5.42 показана диаграмма класса для рассматриваемой здесь реализации.

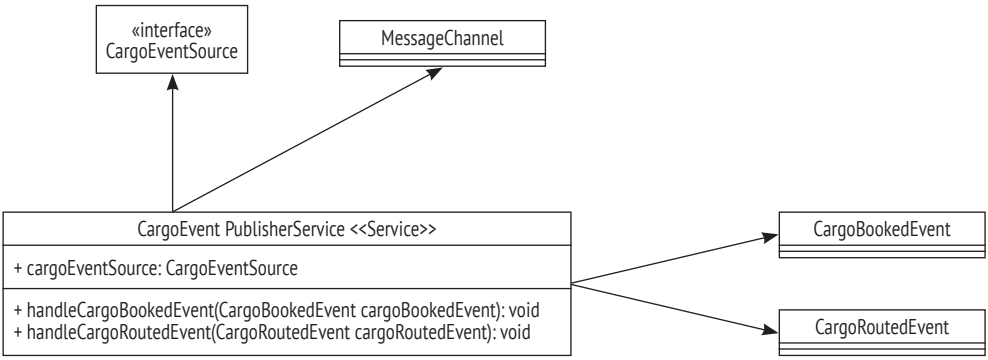

Рис. 5.42 ❖ Диаграмма класса для реализации публикатора событий

На этом завершается рассмотрение реализации исходящих сервисов, сервисов предметной области (домена) и проекта Cargo Tracker как приложения на основе микросервисов, использующего принципы предметно-ориентированного проектирования и платформу Spring.

Итоговый обзор реализации

Теперь мы располагаем полноценной реализацией предметно-ориентированного проектирования в приложении Cargo Tracker на основе микросервисов с различными артефактами предметно-ориентированного проектирования, реализованными с использованием соответствующих проектов, доступных на платформе Spring.

Общая обзорная схема этой реализации показана на рис. 5.43.

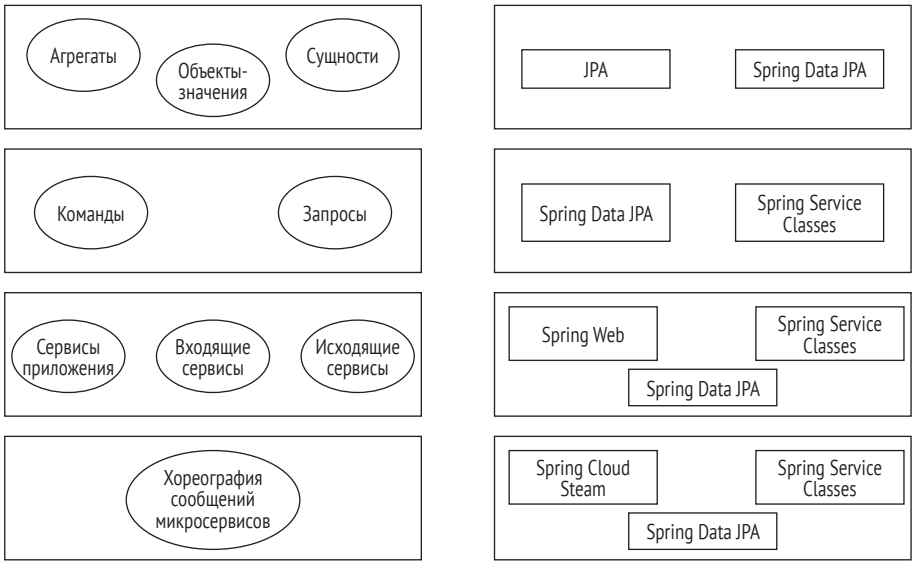

Рис. 5.43 ❖ Общая схема реализации артефактов предметно-ориентированного проектирования с использованием платформы Spring Boot

Резюме

Краткое обобщение содержимого этой главы:

- в начале главы была приведена подробная информация о платформе Spring и о разнообразных функциональных возможностях, предоставляемых этой платформой;
- было принято решение об использовании некоторого подмножества проектов (Spring Boot, Spring Web, Spring Cloud Stream, Spring Data) из полного комплекта платформы Spring для создания приложения Cargo Tracker на основе микросервисов;
- были рассмотрены все подробности процесса разработки и реализации различных артефактов предметно-ориентированного проектирования – сначала модели предметной области (домена), затем сервисов модели предметной области (домена) с использованием выбранных технологий и методик, доступных на платформе Spring.

Глава 6

Проект Cargo Tracker: рабочая среда Axon

К настоящему моменту реализованы три варианта проекта Cargo Tracker:

- реализация предметно-ориентированного проектирования на основе монолитной архитектуры с использованием платформы Jakarta EE;
- реализация предметно-ориентированного проектирования на основе архитектуры микросервисов с использованием платформы Eclipse MicroProfile;
- реализация предметно-ориентированного проектирования на основе архитектуры микросервисов с использованием платформы Spring Boot.

Рассматриваемый в этой главе заключительный вариант реализации предметно-ориентированного проектирования основан на архитектурном шаблоне управляемых событиями микросервисов с использованием следующих инструментальных средств:

- чистой рабочей среды (framework) шаблона ES (Event Sourcing – источники событий);
- чистой методики разделения ответственности команд и запросов CQRS (Command/Query Responsibility Segregation).

Реализация этого варианта будет выполнена с использованием рабочей среды Axon. Axon – это одна из нескольких рабочих сред, доступных в пространстве Enterprise Java, предлагающая готовое к применению «из коробки», стабильное, полноценное, богатое функциональными возможностями решение для реализации архитектуры на основе CQRS/ES.

Использование чистой рабочей среды CQRS/ES, такой как Axon, требует коренных изменений в мыслительном процессе при разработке приложений. Каждый аспект состояния приложения, т. е. формирование состояния, изменение состояния или запросы состояния связан непосредственно с событиями (Events). Такой подход совершенно отличается от обычного метода создания приложений. Основным объектом, представляющим состояние различных ограниченных контекстов приложения, является агрегат (Aggregate), поэтому дальнейшее обсуждение будет в основном связано с состоянием агрегата.

Прежде чем перейти непосредственно к реализации, необходимо более подробно рассмотреть шаблоны Event Sourcing (источники событий) и CQRS, а также методики создания таких приложений. Кроме того, будут определены отличия от предыдущих вариантов реализации проекта.

Шаблон Event Sourcing

Шаблон Event Sourcing (источники событий) применяет собственную, отличную от прочих методику сохранения и извлечения состояний приложения и публикации изменений состояния в различных ограниченных контекстах приложения.

Прежде чем начать подробное изучение шаблона Event Sourcing (ES), рассмотрим обычную методику обработки и сопровождения состояний.

В большинстве обычных приложений используется методика Domain Sourcing или State Sourcing для сохранения и извлечения состояния агрегата. Концепция Domain Sourcing состоит в том, что мы формируем, изменяем и/или запрашиваем состояние агрегата, используя для этого обычный механизм хранения данных (например, реляционные базы данных, базы данных типа NoSQL). Как только устанавливается персистентность состояния агрегата, соответствующее событие публикуется в брокере сообщений. Все варианты реализации в предыдущих главах были основаны на методике Domain Sourcing, показанной на рис. 6.1.

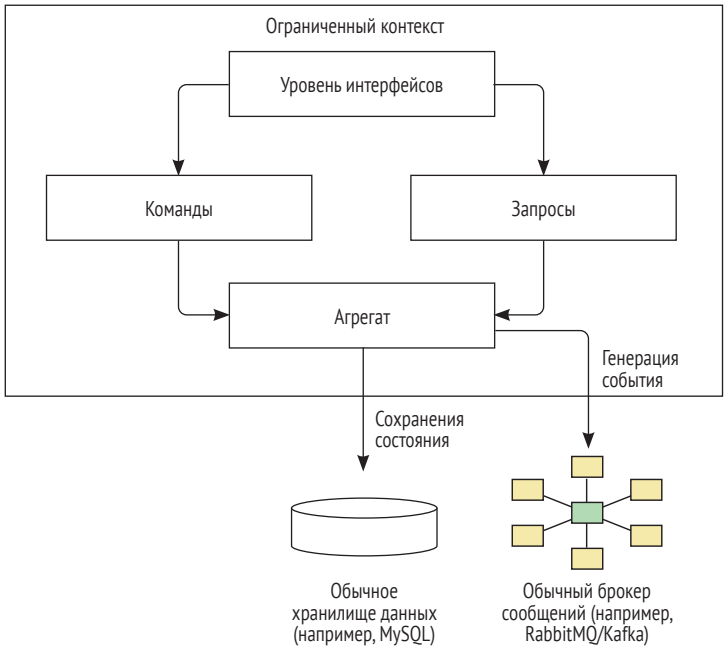

Рис. 6.1 ❖ Приложения Domain Sourcing с сохранением состояния в обычных хранилищах данных

Приложения на основе методики Domain Sourcing весьма просты в эксплуатации, поскольку используют обычные механизмы сохранения и извлечения состояния. Состояние агрегата в различных ограниченных контекстах сохраняется как есть, когда выполняется какая-либо операция в этом агрегате, например, при заказе нового груза создается новый объект Cargo, а подробная

информация о новом грузе сохраняется в соответствующей таблице CARGO базы данных (в нашем случае это база данных Schema в ограниченном контексте Booking). Генерируется событие New Cargo Booked (Заказан новый груз), передающееся в обычный брокер сообщений, в котором может оформить подписку любой другой ограниченный контекст. Для публикации этих событий используется специально выделенный брокер сообщений.

С другой стороны, шаблон Event Sourcing работает исключительно с событиями, которые возникают в агрегатах. Каждое изменение состояния агрегата перехватывается как событие, и персистентность обеспечивается только для этого события, а не для целого агрегата. Экземпляр агрегата представляет собой полезную нагрузку.

Необходимо еще раз подчеркнуть, что сохраняется только событие, а не целый агрегат.

Рассмотрим этот подход подробнее на примере, показанном на рис. 6.2.

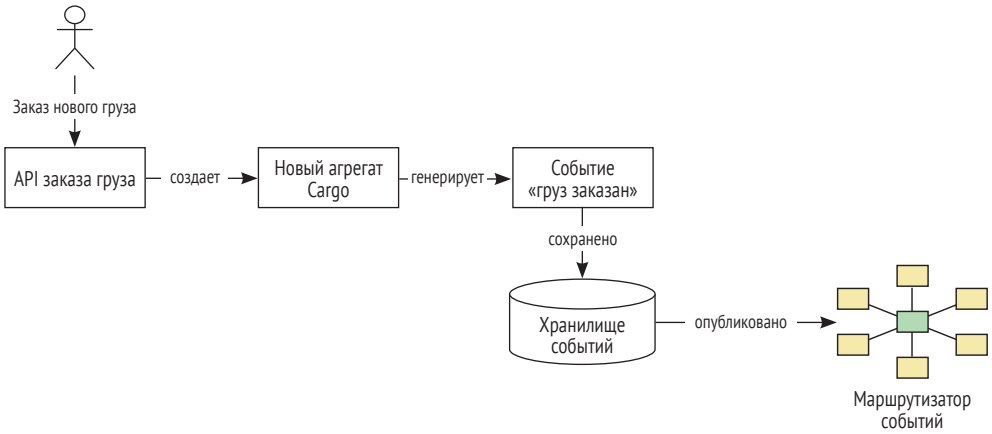

Рис. 6.2 ❖ Пример обработки заказа груза с использованием шаблона Event Sourcing

Как показано на рис. 6.2, в конце операции Book New Cargo (Заказ нового груза) обеспечивается только персистентность события Cargo Booked Event (Груз заказан), а не всего экземпляра агрегата Cargo. Это событие сохраняется в специально созданном для этой цели хранилище событий (Event Store). Кроме того, что это хранилище событий функционирует как механизм обеспечения персистентности для событий, оно также выполняет вторую функцию как маршрутизатор событий (Event Router), т. е. должно сделать сохраненное событие доступным для заинтересованных подписчиков.

При необходимости обновления состояния агрегата также используется абсолютно другая методика, шаги которой описаны ниже:

- из хранилища событий необходимо загрузить набор событий, которые произошли в этом конкретном экземпляре агрегата;
- этот набор воспроизводится в том же экземпляре агрегата для получения его текущего состояния;

- состояние агрегата обновляется (без обеспечения персистентности) на основе выполненной операции;
- обеспечивается персистентность (т. е. сохранение) только обновленного события.

Рассмотрим подробнее и эту методику на примере, показанном на рис. 6.3.

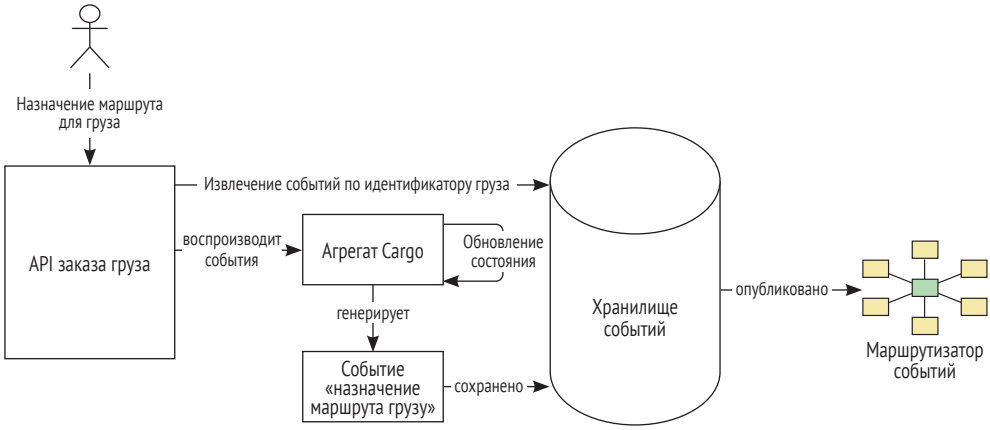

Рис. 6.3 ❖ Пример обработки назначения маршрута для груза с использованием шаблона Event Sourcing

Как показано на рис. 6.3, когда необходимо назначить маршрут доставки груза, в первую очередь извлекается набор событий, которые произошли для конкретного груза, определяемого по его идентификатору (BookingId), затем воспроизводятся события, произошедшие ранее момента (даты) создания соответствующего экземпляра агрегата Cargo. После этого обновляется агрегат Cargo, т. е. добавляется предлагаемый маршрут, наконец публикуется только событие Cargo Routed (маршрут назначен). Еще раз следует отметить, что это – методика, абсолютно отличающаяся от обычной методики, применяемой в приложениях для изменения состояния.

На рис. 6.4 схематически изображены записи в хранилище событий после завершения двух описанных выше операций в агрегате Cargo.

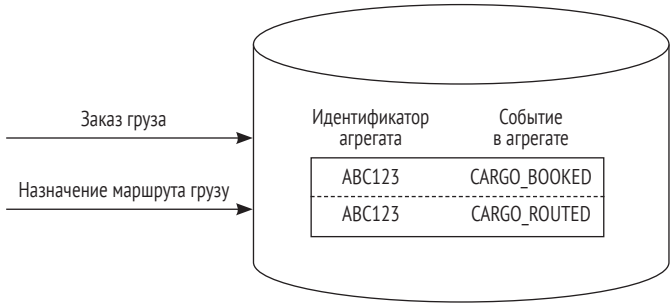

Рис. 6.4 ❖ Хранилище данных о событиях после двух операций в агрегате Cargo

Шаблон Event Sourcing поддерживает особенный способ управления состоянием агрегата в ограниченном контексте, используя для этого методику, основанную исключительно на событиях.

Но каким образом генерировать эти события? Если обеспечивать персистентность только событий, то как получить состояние агрегата? Именно здесь начинает действовать методика разделения ответственности команд и запросов CQRS (Command/Query Responsibility Segregation).

Шаблон Event Sourcing в основном применяется в сочетании с методикой CQRS, при этом область команд используется для генерации событий агрегата, а область запросов – для запросов состояния агрегата.

Методика CQRS

Методика (или принцип) разделения ответственности команд и запросов CQRS (Command/Query Responsibility Segregation), по существу, представляет собой шаблон разработки приложений, который обеспечивает разделение операций, обновляющих состояние, и операций, запрашивающих состояние.

Вообще говоря, методика CQRS определяет следующие принципы работы:

- команды используются для обновления состояния различных объектов приложения (агрегата в ограниченном контексте);
- запросы используются для запросов состояния различных объектов приложения (агрегата в ограниченном контексте).

В предыдущих главах уже описывались реализации некоторых вариантов шаблона CQRS с использованием платформ Jakarta EE, Spring Boot и Eclipse MicroProfile.

На рис. 6.5 показана схема этих предыдущих реализаций CQRS.

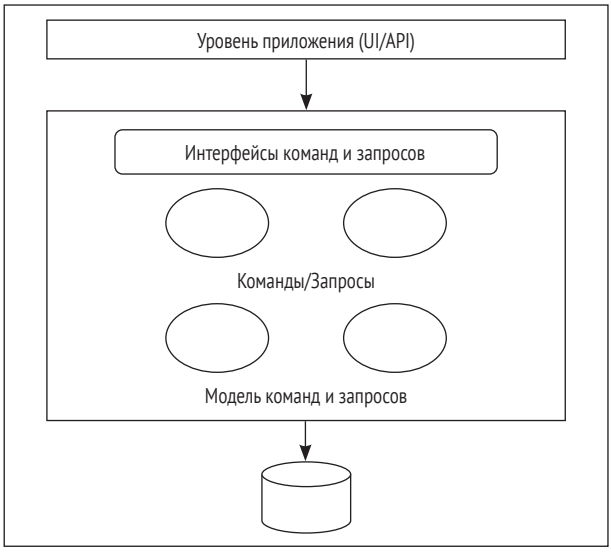

Рис. 6.5 ❖ Схема предыдущих реализаций CQRS

Рассматриваемые ранее реализации были основаны на подходе с совместным использованием, т. е. команды и запросы совместно использовали общую модель (например, агрегат Cargo сам обрабатывал команды и обслуживал запросы). Использовался шаблон Domain Sourcing, т. е. состояние сохранялось и извлекалось в форме «как есть» в обычной базе данных.

При необходимости использования методики CQRS в сочетании с шаблоном Event Sourcing все происходит немного по-другому. При таком подходе команды и запросы имеют:

- отдельные модели;
- отдельные потоки кода;
- отдельные интерфейсы;
- отдельные логические процессы;
- отдельные хранилища, обеспечивающие персистентность.

Схема этого подхода изображена на рис. 6.6.

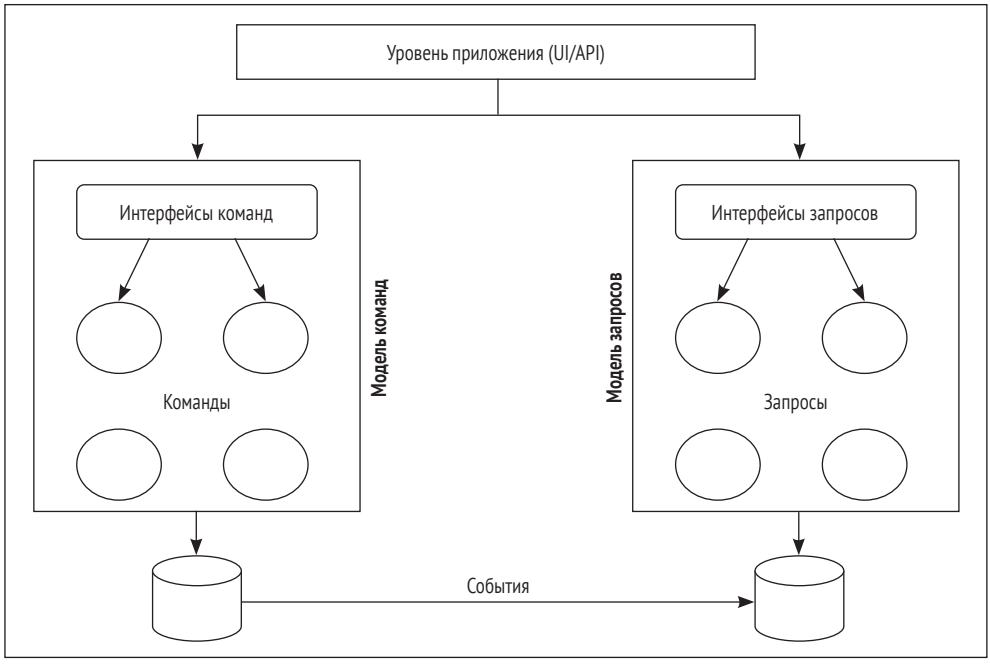

Рис. 6.6 ❖ Разделенная модель с использованием CQRS и ES

Как можно видеть на рис. 6.6, в ограниченном контексте команды и запросы имеют собственные наборы интерфейсов, моделей, процессов и хранилищ. Сторона (область) команд обрабатывает команды для изменения состояния агрегата. Результатом являются события (Events), которые сохраняются (обеспечивается их персистентность), и на них подписывается сторона (область) запросов для обновления модели чтения. Модель чтения (Read Model) – это проекция состояния приложения, ориентированная на специализированную аудиторию со специализированными требованиями к информации. На эти события могут подписаться другие ограниченные контексты.

На рис. 6.7 показано объединение шаблонов CQRS и ES.

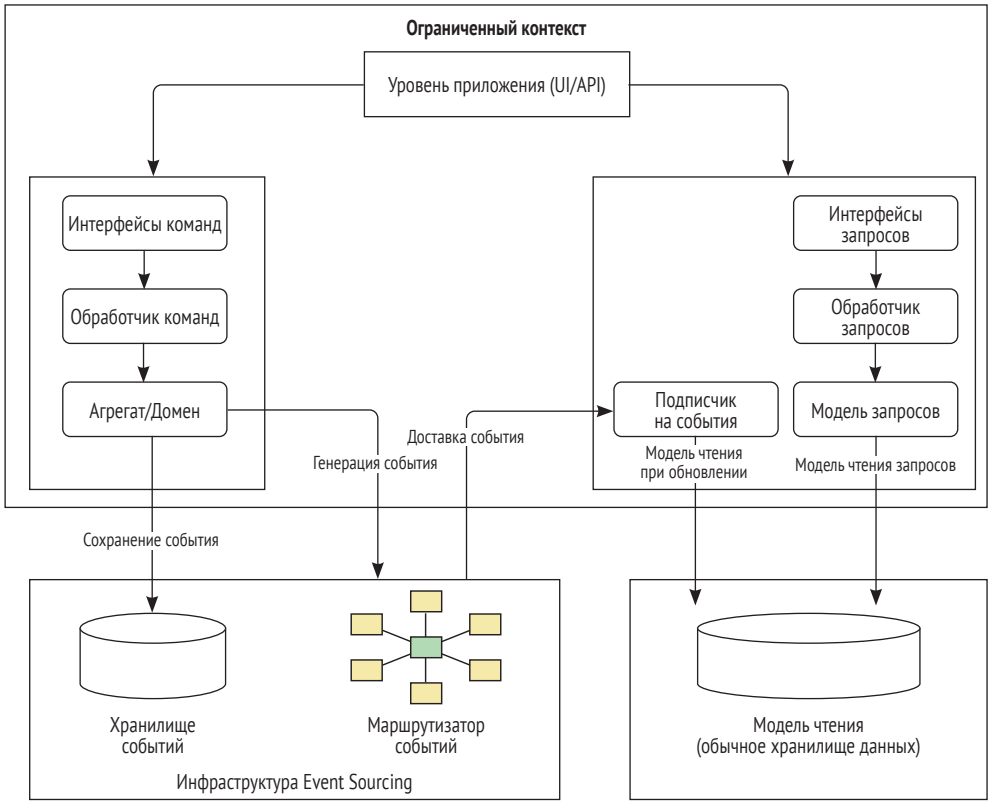

Рис. 6.7 ❖ Методика CQRS в сочетании с шаблоном Event Sourcing

Подведем итоги, обобщив принципы создания приложений с использованием CQRS и Event Sourcing:

- события являются гражданами первого класса (first-class citizens);
- используется модель команд, которая обновляет состояние агрегатов и генерирует события;
- сохраняются события, а не состояние приложения в целом. Для сохранения создается специализированное хранилище событий (Event Store). Кроме того, хранилище событий дублируется в форме маршрутизатора событий (Event Router), который делает сохраненные события доступными для заинтересованных подписчиков;
- модель чтения (Read Model)/проекция состояния агрегата представлена через модель запросов (Query Model), которая обновляется по подписке на события изменения состояния.

Программы, использующие такой шаблон, специально предназначены для создания управляемых событиями приложений на основе микросервисов.

Прежде чем перейти непосредственно к реализации, необходимо ознакомиться с кратким вводным описанием рабочей среды Axon.

Рабочая среда Axon

Рабочая среда Axon появилась в 2010 году как программный комплекс CQRS/ES с открытым исходным кодом.

В последние годы наблюдается быстрое развитие этой рабочей среды и в дополнение к ее ядру предлагается функция сервера, которая включает хранилище событий (Event Store) и маршрутизатор событий (Event Router). Ядро рабочей среды Axon в совокупности с этим сервером позволяет абстрагировать сложные инфраструктурные компоненты, требуемые при реализации шаблонов CQRS/ES, и помогает разработчикам корпоративных приложения сосредоточиться исключительно на бизнес-логике.

Реализация архитектуры на основе шаблона Event Sourcing чрезвычайно сложна на практике. В настоящее время наблюдается возрастающая тенденция к реализации хранилищ событий с использованием платформ стриминга (потоковых платформ), например Kafka. Недостатком такого подхода является значительный объем специализированных трудозатрат при реализации функций Event Sourcing, которые не предоставляют эти платформы стриминга (поскольку они именно платформы стриминга, а не платформы Event Sourcing). Axon является лидером в этой области и набор функциональных возможностей этой рабочей среды существенно облегчает реализацию шаблонов CQRS/ES в приложениях.

Превосходным дополнением является возможность применения предметно-ориентированного проектирования как основополагающего структурного блока для создания приложений. С учетом того что в последнее время промышленные предприятия весьма активно переходят на архитектурный стиль микросервисов, Axon со своей методикой объединения предметно-ориентированного проектирования и шаблонов CQRS/ES предоставляет надежное и полнофункциональное решение для создания микросервисов, управляемых событиями и специализированных для конкретных клиентов.

Компоненты рабочей среды Axon

На самом высоком уровне Axon предоставляет следующие компоненты:

- Axon Framework, Domain Model – ядро рабочей среды, помогающее создать модель предметной области (домена) на основе предметно-ориентированного проектирования и шаблонов Event Sourcing и CQRS;
- Axon Framework, Dispatch Model – логическую инфраструктуру для поддержки упомянутой выше модели предметной области (домена), т. е. для маршрутизации и координации команд и запросов, имеющих дело с состоянием этой предметной области (домена);
- Axon Server – физическую инфраструктуру для поддержки упомянутых выше моделей предметной области (домена) и координации.

Схема компонентов рабочей среды Axon показана на рис. 6.8.

В любом случае имеется возможность выбора внешней инфраструктуры вместо сервера Axon, упомянутого выше, но это приведет к необходимости реализации набора функциональных возможностей, которые сервер Axon предоставляет готовыми к непосредственному использованию «прямо из коробки».

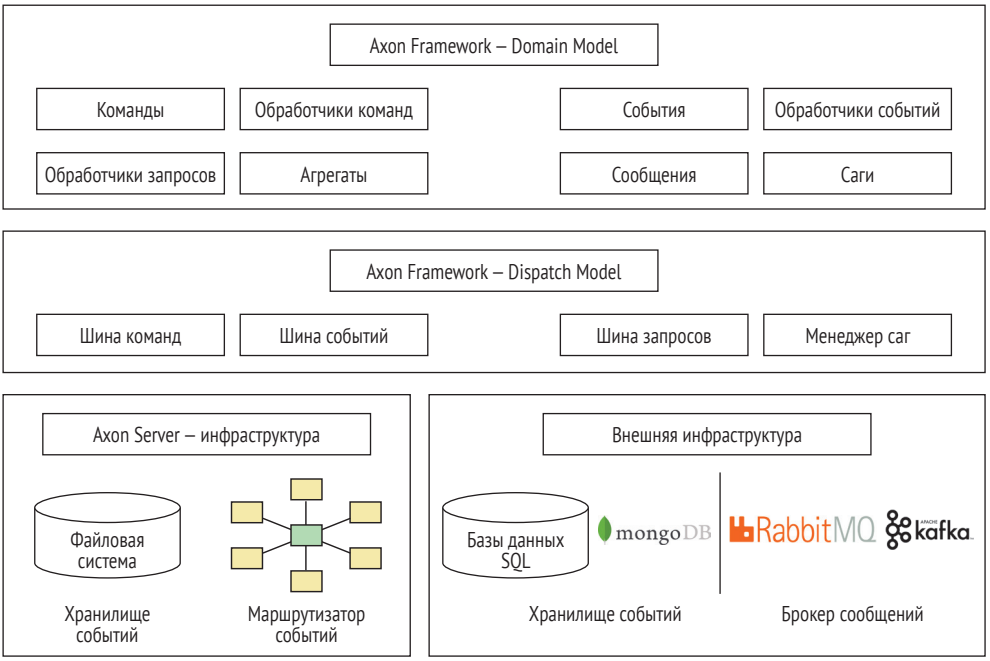

Рис. 6.8 ❖ Компоненты рабочей среды Axon

В следующем разделе приведен весьма краткий обзор компонентов рабочей среды Axon. В процессе реализации модели предметной области (домена) для приложения Cargo Tracker мы будем более подробно их рассматривать, поэтому сейчас вполне достаточно ознакомиться со следующим разделом, чтобы получить общее представление об этих компонентах.

Компоненты предметной области (домена) Axon Framework

Агрегаты

Как основу модели предметной области (домена) любого ограниченного контекста Axon предоставляет поддержку объектов первого класса для определения и разработки агрегатов предметно-ориентированного проектирования. В рабочей среде Axon агрегаты реализуются как обычные простые старые объекты Java (POJO), которые содержат состояние и методы для изменения этого состояния. Объекты POJO помечаются стандартной аннотацией `@Aggregate`, чтобы определить их как агрегаты.

Кроме того, рабочая среда Axon предоставляет поддержку идентификации агрегатов и механизма обработки команд (изменения состояния) в агрегатах, а также загрузку этих агрегатов из хранилища событий (в соответствии с шаблоном Event Sourcing). Такая поддержка обеспечивается посредством использования специализированных стандартных аннотаций `@AggregateIdentifier`, `@CommandHandler`, `@EventSourcingHandler`.

Команды и обработчики команд

В командах передается инструкция об изменении состояния агрегатов в различных ограниченных контекстах.

Axon предоставляет поддержку объектов первого класса для обработки команд с помощью обработчиков команд. Обработчики команд – это подпрограммы, которые размещены в агрегате. Они принимают заданную команду, т. е. инструкцию об изменении состояния, как основные входные данные. Действительные классы команд реализуются как обычные простые старые объекты Java (POJO), но поддержка обработчика команд обеспечивается с помощью стандартных аннотаций `@CommandHandler`, размещаемых в агрегате. Axon также поддерживает размещение этих команд во внешнем классе (External Command Handler), если это необходимо. Обработчики команд также отвечают за генерацию событий предметной области (домена) и за делегирование этих событий в инфраструктуру хранилища событий/маршрутизатора событий рабочей среды Axon.

События и обработчики событий

Обработка команд в агрегатах всегда приводит к генерации событий. События сообщают об изменении состояния агрегата в ограниченном контексте заинтересованным подписчикам. Сами классы событий реализуются как обычные простые старые объекты Java (POJO), которые не требуют каких-либо специальных аннотаций. Агрегаты используют методы жизненного цикла, которые Axon предоставляет для передачи событий в хранилище событий, затем в маршрутизатор событий после обработки команды.

Потребление (прием) событий обрабатывается с помощью обработчиков событий, которые подписываются на события, в которых они заинтересованы. Axon предоставляет стандартную аннотацию `@EventHandler`, размещаемую в подпрограммах внутри обычных простых старых объектов Java (POJO), которые обеспечивают прием и последующую обработку событий.

Обработчики запросов

В запросах передаются инструкции по извлечению состояния агрегатов в ограниченных контекстах. В рабочей среде Axon запросы обрабатываются с помощью обработчиков запросов, обозначаемых аннотацией `@QueryHandler`, которая размещается в обычных простых старых объектах Java (POJO). Обработчики запросов для извлечения состояния агрегата используют модель чтения (Read Model)/проекцию (Projection) хранилища данных. Для выполнения внешних запросов (requests) применяются обычные программные среды (например, Spring Data, JPA).

Саги

Рабочая среда Axon обеспечивает поддержку объектов первого класса и для саг на основе хореографии, и для саг на основе оркестровки. Следует кратко напомнить, что саги на основе хореографии полагаются на события, генерируемые и потребляемые через подписку различными ограниченными контекстами, которые являются участниками саги. С другой стороны, саги на основе

оркестровки полагаются на централизованный компонент, отвечающий за координацию событий между различными ограниченными контекстами, которые являются участниками саги. Саги на основе хореографии формируются с помощью обычных обработчиков событий, предоставляемых рабочей средой Axon. Кроме того, рабочая среда Axon предоставляет полноценную реализацию для поддержки саг на основе оркестровки. Реализации саг включают следующие элементы:

- управление жизненным циклом (запуск и завершение саг с помощью соответствующих аннотаций управления сагами Saga Managers);
- обработку событий (с помощью аннотации @SagaEventHandler);
- хранилище состояния саги с поддержкой нескольких различных реализаций (JDBC, Mongo и т.п.);
- управление связями между несколькими сервисами (Association Management);
- обработку предельного срока завершения саги (Deadline Handling).

На этом завершается рассмотрение компонентов модели предметной области (домена) в рабочей среде Axon. В следующем разделе кратко описываются компоненты модели регулирования и координации в рабочей среде Axon.

Компоненты модели регулирования и координации Axon Framework

Модель регулирования и координации (Dispatch Model) рабочей среды Axon важно хорошо понимать при создании приложений на основе Axon. Следует напомнить, что любой ограниченный контекст участвует в четырех типах операций:

- обработке команд для изменения состояния;
- обработке запросов для извлечения состояния;
- публикации и потребление событий;
- сагах.

Модель регулирования и координации рабочей среды Axon предоставляет необходимую инфраструктуру, позволяющую ограниченным контекстам участвовать в перечисленных операциях. Например, при передаче команды в ограниченный контекст именно модель регулирования и координации обеспечивает корректное перенаправление принятой команды в соответствующий обработчик команды в этом ограниченном контексте.

Рассмотрим модель регулирования и координации более подробно.

Шина команд

Команды, передаваемые в ограниченный контекст, должны обрабатываться обработчиками команд. Шина команд и шлюз команд обеспечивают регулирование направления команд в соответствующие обработчики команд:

- шину команд (Command Bus) – компонент инфраструктуры Axon, выполняющий маршрутизацию команд в соответствующий обработчик `CommandHandler`;
- шлюз команд (Command Gateway) – компонент (утилиту) инфраструктуры Axon, представляющий собой обертку для шины команд. Использова-

ние шины команд требует написания повторяющегося кода для каждой команды регулирования (например, создание `CommandMessage`, подпрограммы `CommandCallback`). Применение шлюза команд помогает исключить большой объем повторяющегося кода. Обсуждение этого вопроса будет продолжено в последующих разделах этой главы.

Рабочая среда Axon предоставляет несколько вариантов реализации шины команд `CommandBus`:

- SimpleCommandBus;
- AxonServerCommandBus;
- AsynchronousCommandBus;
- DisruptorCommandBus;
- DistributedCommandBus;
- RecordingCommandBus.

Для рассматриваемого здесь варианта реализации будет использоваться шина команд AxonServerCommandBus с применением Axon Server как механизма регулирования и координации разнообразных команд.

Общая схема этого варианта реализации показана на рис. 6.9.

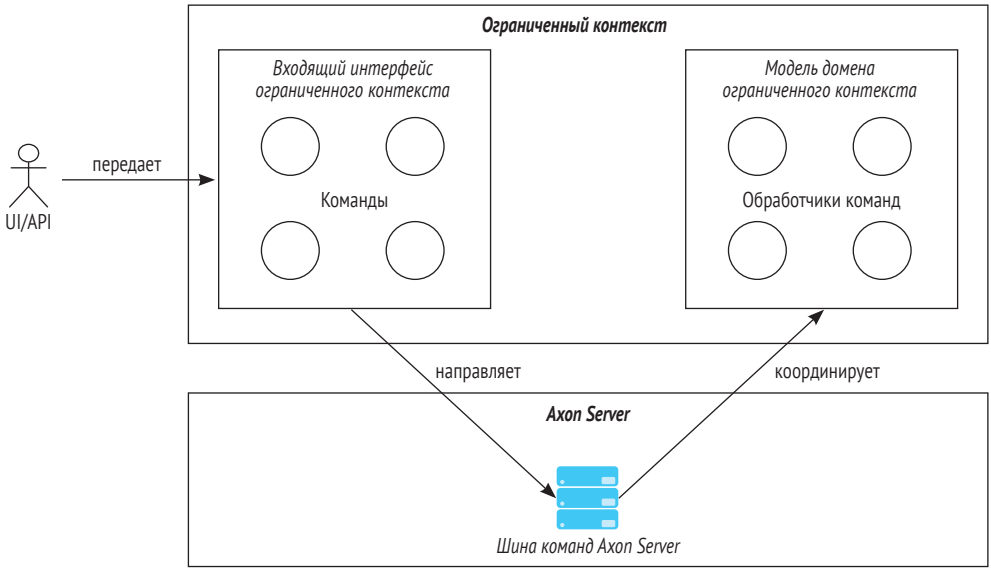

Рис. 6.9 ❖ Реализация шины команд с использованием Axon Server

Шина запросов

Как и команды, запросы, передаваемые в ограниченный контекст, должны обрабатываться обработчиками запросов. Шина запросов и шлюз запросов обеспечивают регулирование направления запросов в соответствующие обработчики запросов:

- шину запросов (Query Bus) – компонент инфраструктуры Axon, выполняющий маршрутизацию команд в соответствующий обработчик `QueryHandler`;

- шлюз запросов (Query Gateway) – компонент (утилиту) инфраструктуры Axon, представляющий собой обертку для шины запросов. Использование шины запросов помогает исключить большой объем повторяющегося кода.

Рабочая среда Axon предоставляет несколько вариантов реализации шины запросов `QueryBus`:

- SimpleQueryBus;
- AxonServerQueryBus.

Для рассматриваемого здесь варианта реализации будет использоваться шина запросов AxonServerQueryBus с применением Axon Server как механизма регулирования и координации разнообразных запросов.

Общая схема этого варианта реализации показана на рис. 6.10.

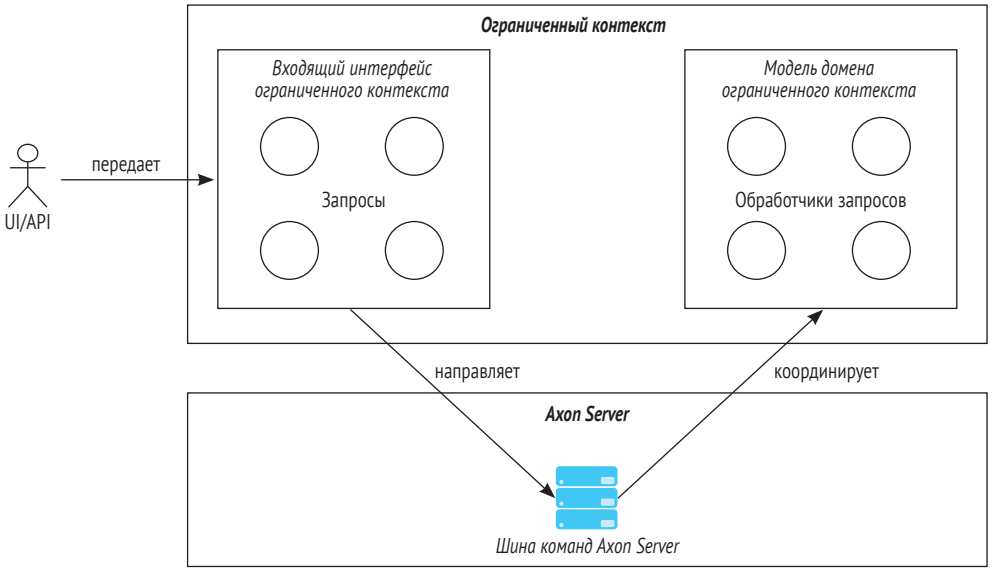

Рис. 6.10 ❖ Реализация шины запросов с использованием Axon Server

Шина событий

Шина событий (Event Bus) – это механизм, обеспечивающий прием событий из обработчиков команд и перенаправление их в соответствующие обработчики событий, которыми могут быть любые другие ограниченные контексты, заинтересованные в каком-либо конкретном событии. Рабочая среда Axon предоставляет три варианта реализации шины событий:

- AxonServerEventStore;
- EmbeddedEventStore;
- SimpleEventBus.

Для рассматриваемого здесь варианта реализации будет использоваться шина событий AxonServerEventStore. На рис. 6.11 показан механизм шины событий в рабочей среде Axon.

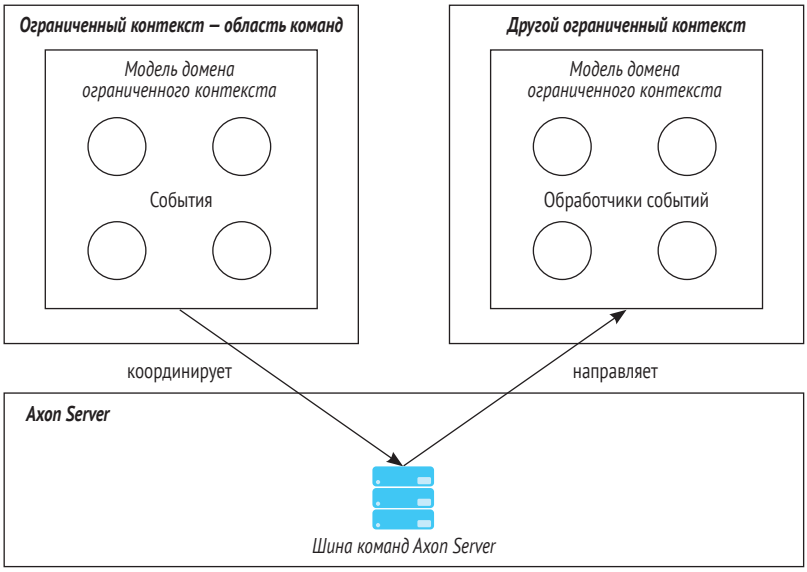

Рис. 6.11 ❖ Реализация шины событий с использованием Axon Server

Саги

Как уже было отмечено выше, рабочая среда Axon предоставляет поддержку и для саг на основе оркестровки, и для саг на основе хореографии. Реализация саг на основе хореографии проста и очевидна в том смысле, что ограниченные контексты, участвующие в конкретной саге, будут непосредственно генерировать события и подписываться на них точно так же, как и при обычной обработке событий.

С другой стороны, в сагах на основе оркестровки координация жизненного цикла обеспечивается централизованным компонентом. Этот централизованный компонент отвечает за создание саги, за координацию потоков между различными ограниченными контекстами, участвующими в саге, и за корректное завершение самой саги. Рабочая среда Axon предоставляет компонент SagaManager для этой цели. Саги на основе оркестровки также требуют сохранения состояния в хранилищах данных и извлечения экземпляров саг. Существуют разнообразные реализации таких хранилищ, поддерживаемые рабочей средой Axon (JPA, БД в памяти, JDBC и Mongo, Axon Server). В рассматриваемом здесь варианте реализации будет использоваться хранилище саг, предоставляемое сервером Axon Server. Кроме того, рабочая среда Axon применяет разумные значения параметров по умолчанию и автоматически конфигурирует компонент SagaManager и сервер Axon Server как механизм сохранения состояния при создании саги.

На рис. 6.12 показан механизм саги на основе оркестровки, используемый в рассматриваемой здесь реализации.

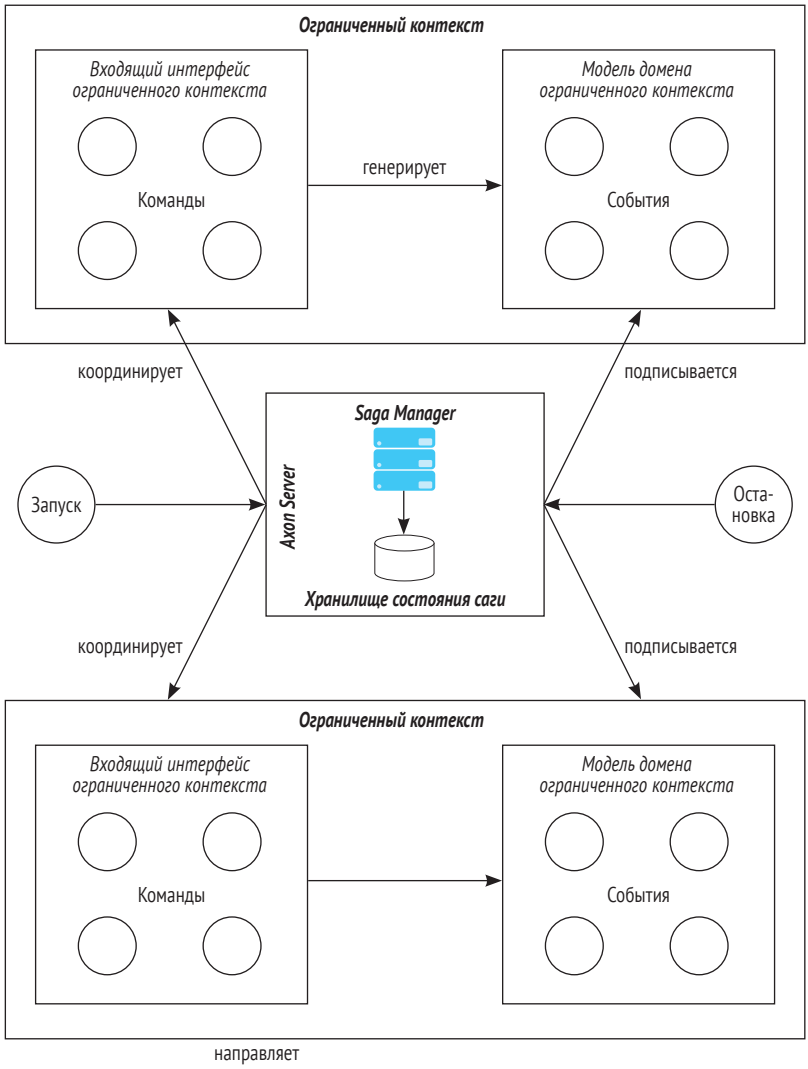

Рис. 6.12 ❖ Механизм саги на основе оркестровки в рабочей среде Axon

Компоненты инфраструктуры Axon: Axon Server

Сервер Axon Server предоставляет физическую инфраструктуру, необходимую для поддержки модели регулирования и координации. В широком смысле Axon Server содержит два основных компонента:

- хранилище событий на основе H2 (используемого для хранения конфигурации) и файловую систему (используемую для хранения данных событий);
- маршрутизатор сообщений для организации потока событий в системе.

Ниже приводится краткое описание функциональных возможностей сервера:

- встроенный маршрутизатор сообщений (Messaging Router) с поддержкой расширенных шаблонов сообщений (Sticky Command Routing, Message Throttling, QoS);
- специализированное по целям хранилище событий с встроенной базой данных (H2);
- возможности высокой доступности и масштабируемости (кластеризация);
- функции управления обеспечением безопасности;
- консоль пользовательского интерфейса (UI Console);
- функции мониторинга и метрики;
- функции управления данными (резервное копирование, точная настройка, управление версиями).

Axon Server формируется с использованием платформы Spring Boot и распространяется как обычный JAR-файл (текущая версия axonserver-4.1.2). Сервер использует собственную файловую систему на основе механизма хранения как базу данных для сохранения событий. Скачать Axon Server можно на сайте www.axoniq.io.

Установка и запуск сервера выполняется так же просто, как запуск обычного JAR-файла. В листинге 6.1 показан пример команды запуска.

Листинг 6.1 ❖ Команда запуска Axon Server

```
java -jar axonserver-4.1.2.jar
```

Эта команда активизирует Axon Server, а к его консоли можно получить доступ по адресу `http://localhost:8024`. На рис. 6.13 показана панель управления как часть пользовательского интерфейса в консоли, предоставляемой Axon Server.

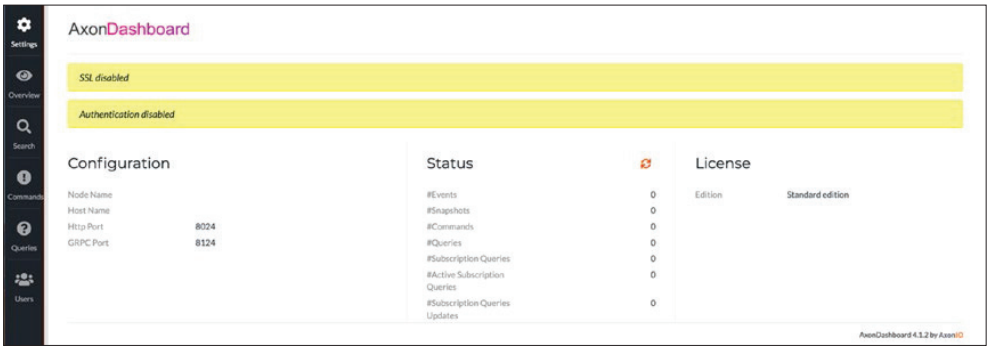

Рис. 6.13 ❖ Консоль Axon Server

Пользовательская консоль предоставляет функциональные возможности для мониторинга и управления сервером Axon Server. Ниже приведен краткий обзор этих функциональных возможностей. При рассмотрении примера реализации в этой главе эти функциональные возможности будут описываться более подробно. Сейчас вполне достаточно ознакомиться с общими характеристиками:

- **Settings** (Настройки) – это начальная страница панели управления сервера (Server Dashboard). Она содержит всю подробную информацию о конфигурации, о состоянии различных операций, а также подробные данные о лицензии и о системе обеспечения безопасности. Здесь необходимо отметить, что Axon поддерживает HTTP и gRPC как входящие протоколы;
- **Overview** (Общая схема) – на этой странице представлена визуальная графическая схема Axon Server и экземпляров приложений, установивших соединения с сервером. Поскольку в настоящий момент пока еще не создано ни одного приложения, на схеме отображается только основной сервер, как показано на рис. 6.14.

Рис. 6.14 ❖ Графическое представление схемы Axon Server

- **Search** (Поиск) – на этой странице отображается визуальное представление хранилища событий более низкого уровня. Рабочая среда Axon также предоставляет язык запросов для работы с хранилищем событий. Это представление показано на рис. 6.15.

Рис. 6.15 ❖ Страница запросов Axon Query
со специализированным языком Query DSL

Результаты поиска (по запросам) показаны на рис. 6.16.

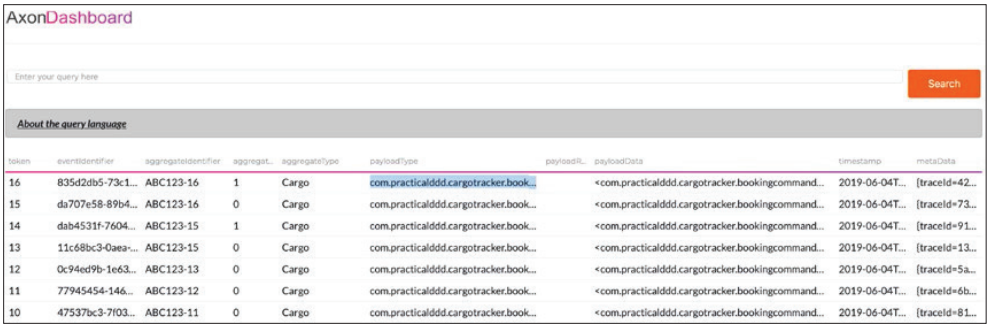

Рис. 6.16 ❖ Результаты поиска по запросу в странице консоли Axon Query

- **Users** (Пользователи) – эта страница предоставляет функции добавления и удаления пользователей с назначением им соответствующих ролей (ADMIN/USER) для доступа к серверу Axon Server. Эта страница показана на рис. 6.17.

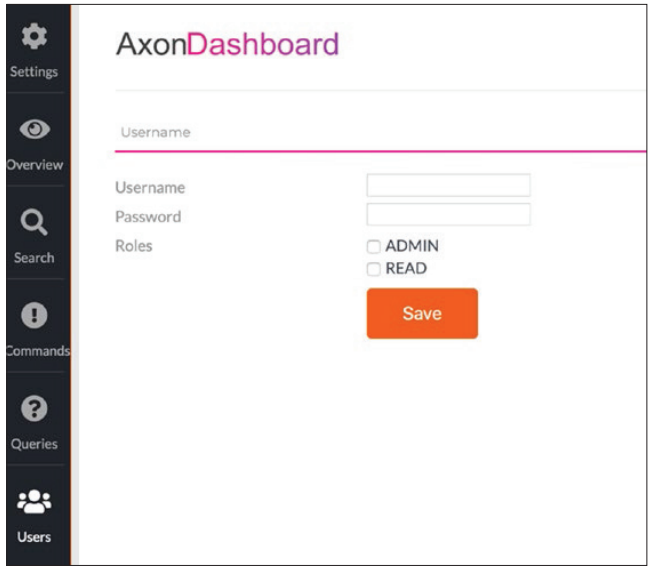

Рис. 6.17 ❖ Администрирование пользователей Axon Query

Подведем итоги обзора:

- Axon предлагает чистую реализацию для приложений, в которых необходимо использовать CQRS/Event Sourcing, микросервисы, управляемые событиями и предметно-ориентированное проектирование как основные архитектурные шаблоны;
- Axon предоставляет модель предметной области (домена) и модель регулирования и координации (Axon Framework), а также поддержку хра-

нилища событий и инфраструктуры маршрутизации сообщений (Axon Server), помогая создавать приложения на основе CQRS/ES;
- Axon позиционирует события и шаблон Event Sourcing как основополагающие структурные блоки для создания приложений на основе микросервисов, управляемых событиями, с использованием CQRS/ES;
- Axon предоставляет административную консоль для выполнения запросов, обеспечения безопасности и администрирования данных событий.

После изучения функциональных возможностей рабочей среды Axon мы переходим к рассмотрению подробностей реализации приложения Cargo Tracker.

Приложение Cargo Tracker и рабочая среда Axon

Этот вариант реализации приложения Cargo Tracker будет основан на архитектуре микросервисов, управляемых событиями, с использованием ядра рабочей среды Axon (Axon Framework) и инфраструктуры Axon (Axon Server).

Ядро рабочей среды Axon Framework обеспечивает поддержку объектов первого класса для платформы Spring Boot как технологии более низкого уровня для создания и функционирования разнообразных артефактов Axon. Несмотря на то что нет необходимости в обязательном использовании платформы Spring Boot при поддержке, которую обеспечивает Axon, тем не менее процедура конфигурирования компонентов становится чрезвычайно простой при использовании функциональных возможностей конфигурирования платформы Spring Boot.

Ограниченные контексты в Axon

Для рассматриваемых в этой книге реализаций микросервисов применяется методика разделения приложения Cargo Tracker на четыре ограниченных контекста. Каждый ограниченный контекст содержит собственный набор микросервисов. Рассматриваемая в этой главе реализация с использованием рабочей среды Axon не является исключением.

На рис. 6.18 изображены четыре ограниченных контекста в приложении Cargo Tracker.

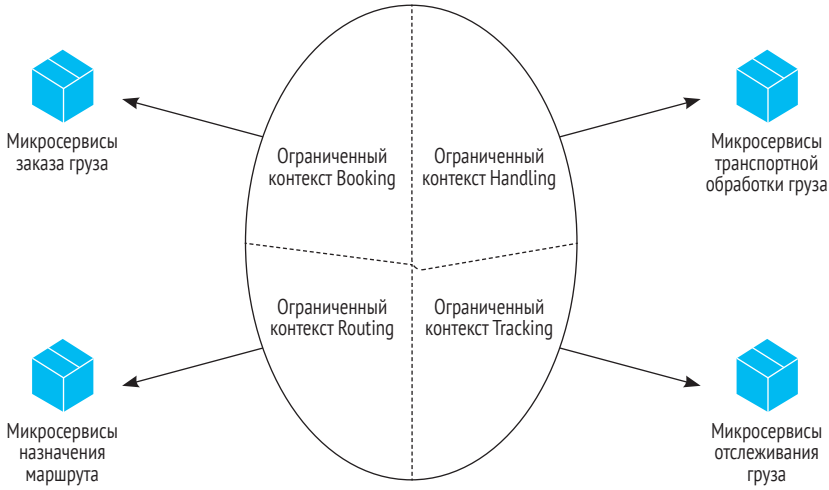

Рис. 6.18 ❖ Ограниченные контексты в приложении Cargo Tracker

Несмотря на то что используется та же самая методика разделения на ограниченные контексты, что и в предыдущих реализациях микросервисов, здесь возникает множество аспектов, абсолютно отличающихся от тех, которые мы рассматривали ранее. В этом примере реализации каждый ограниченный контекст разделяется на область (сторону) команд (Command Side) и область (сторону) запросов (Query Side). Область команд ограниченного контекста обрабатывает любые запросы на изменение состояния этого ограниченного контекста. Область команд также генерирует и публикует изменения состояния агрегата как события. Область запросов каждого ограниченного контекста предоставляет модель чтения (Read Model)/проекцию (Projection) текущего состояния агрегата, используя для этого отдельное хранилище, обеспечивающее персистентность. Эта модель чтения/проекция обновляется областью запросов посредством подписки на события изменения состояния.

Общая схема этой структуры показана на рис. 6.19.

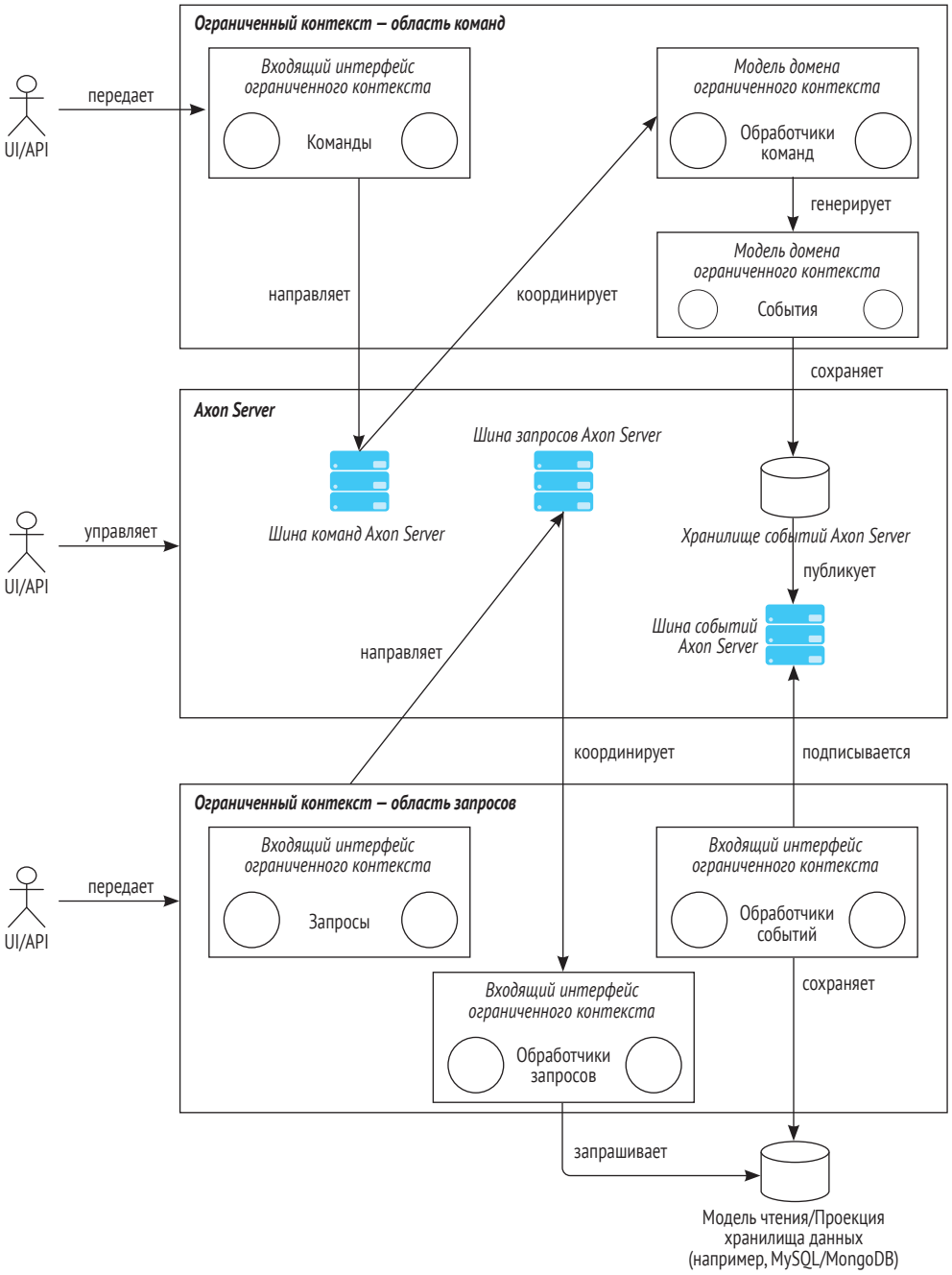

Рис. 6.19 ❖ Ограниченные контексты – область команд и область запросов

Ограниченные контексты: создание артефакта

Каждый ограниченный контекст имеет собственный развертываемый артефакт. Каждый из рассматриваемых здесь ограниченных контекстов содержит набор микросервисов, который может быть разработан, развернут и промасштабирован независимо от прочих. Каждый такой артефакт создается как файл Axon + Spring Boot fat Jar, который содержит все требуемые зависимости и среду времени выполнения, необходимую для независимой работы.

Общая схема артефакта показана на рис. 6.20.

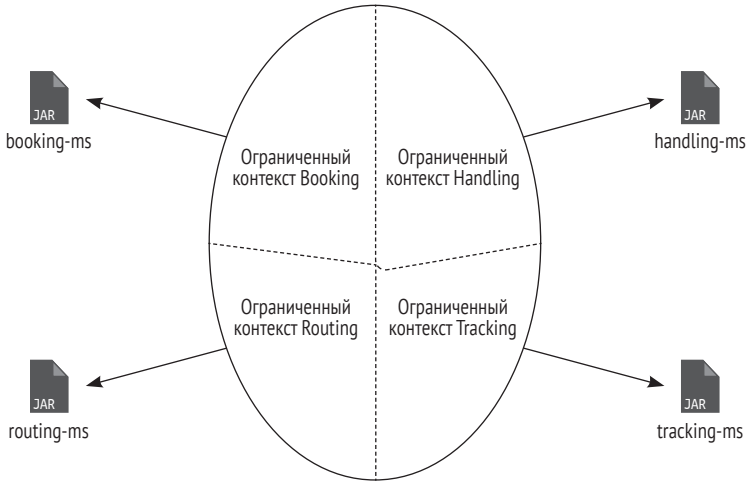

Рис. 6.20 ❖ Ограниченные контексты – отображение в соответствующие артефакты микросервисов

Для начала работы с Axon в первую очередь необходимо создать обычное приложение Spring Boot со следующими зависимостями: spring-web и spring-data-jpa.

На рис. 6.21 показано создание микросервисов заказа груза с использованием проекта Initializr из платформы Spring Boot (start.spring.io).

Проект создан со следующими характеристиками:

- группа – com.practicalddd.cargotracker;
- артефакт – bookingms;
- зависимости – Spring Web Starter, Spring Data JPA.

Axon использует функциональные возможности автоматической конфигурации платформы Spring Boot для конфигурирования компонентов этого приложения. Для обеспечения такой интеграции просто добавляется зависимость `axon-spring-boot-starter` в pom-файл проекта Spring Boot. После того как эта зависимость становится доступной, `axon-spring-boot-starter` автоматически сконфигурирует модель регулирования и координации (Command Bus, Query Bus, Event Bus) и хранилище событий (Event Store).

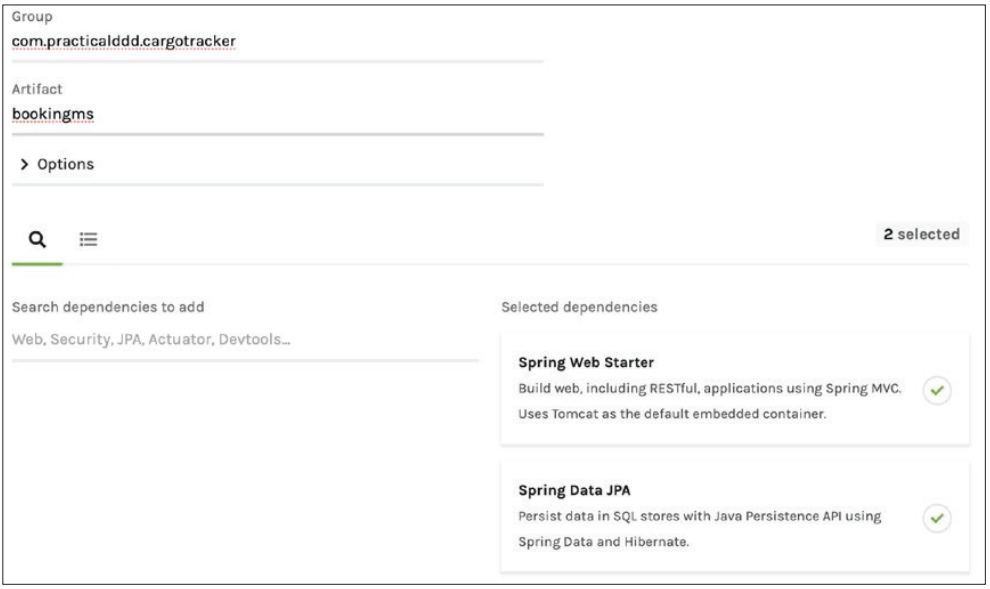

Рис. 6.21 ❖ Проект Spring Boot – микросервисы заказа груза со всеми зависимостями

Добавление этой зависимости показано в листинге 6.2.

Листинг 6.2 ❖ Зависимости для Axon Spring Boot Starter

```
<dependency>
      <groupId>org.axonframework</groupId>
      <artifactId>axon-spring-boot-starter</artifactId>
      <version>4.1.1</version>
</dependency>
```

Axon использует разумные значения параметров по умолчанию для конфигурирования Axon Server как инфраструктуры модели регулирования и координации и хранилища событий. Нет необходимости включать эти параметры в каждый экземпляр конфигурации или в файлы исходного кода приложения Spring Boot. Простое добавление зависимости, показанной в листинге 6.2, автоматически конфигурирует Axon Server как реализацию инфраструктуры модели регулирования и координации, а также как реализацию хранилища событий.

Общая структура приложения Axon Spring Boot показана на рис. 6.22.

Ограниченные контексты: структура пакета

Первый шаг реализации ограниченных контекстов – логическое группирование и объединение различных артефактов Axon в один развертываемый артефакт. Такое логическое группирование предполагает идентификацию структуры пакета, в котором размещаются различные артефакты Axon, для получения общего решения для рассматриваемых здесь ограниченных контекстов.

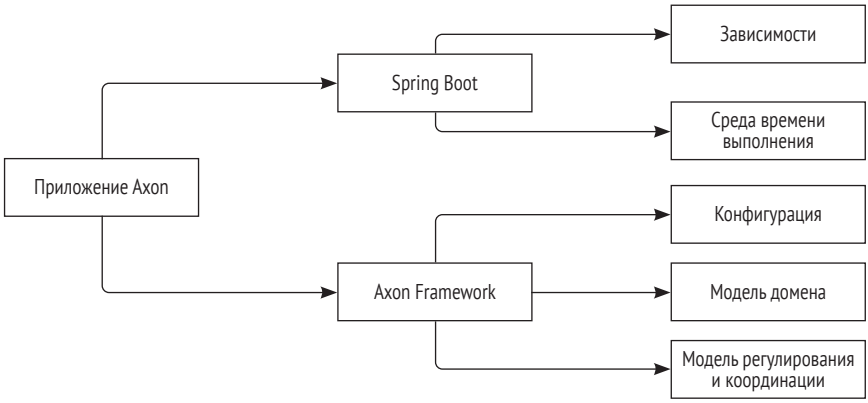

Рис. 6.22 ❖ Общая схема приложения Axon Spring Boot

На самом высоком уровне структура пакета для любого ограниченного контекста (Command Side, Query Side) показана на рис. 6.23. На этой схеме можно видеть, что структура пакета ничем не отличается от структуры в предыдущих реализациях, так как шаблон CQRS/ES полностью соответствует гексагональной архитектуре, которая подробно рассматривалась в главе 2 (рис. 2.9).

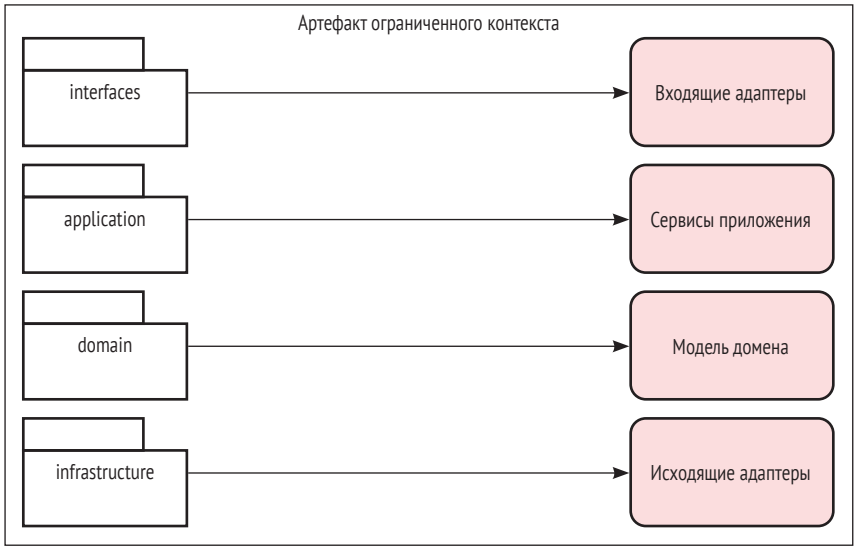

Рис. 6.23 ❖ Ограниченные контексты – структура пакета

Рассмотрим подробнее структуру пакета на примерах ограниченных контекстов области команд и области запросов Booking (заказ груза).

Пакет interfaces

В этот пакет включены все входящие интерфейсы для рассматриваемого ограниченного контекста, классифицированные по протоколам обмена данными.

Главная цель интерфейсов – согласование протокола от имени модели предметной области (домена) (например, REST API, WebSocket, FTP, какой-либо специализированный протокол).

В рассматриваемом здесь примере ограниченный контекст команд Booking (заказ груза) предоставляет REST API для передачи ему команд (например, команд Book Cargo, Update Cargo). Кроме того, ограниченный контекст запросов Booking предоставляет REST API для передачи ему запросов (например, Retrieve Cargo Booking Details, List all Cargos). Эти интерфейсы сгруппированы в пакете *rest*. К интерфейсам также относятся обработчики событий, которые подписываются на различные события, генерируемые рабочей средой Axon. Все обработчики событий собраны в пакете *eventhandlers*. В дополнение к этим двум пакетам пакет интерфейсов также содержит пакет *transform*, который используется для преобразования входных данных ресурсов API и событий в соответствующий формат (модель) команд и запросов.

Структура пакета interfaces показана на рис. 6.24. Она одинакова вне зависимости от того, принадлежит ли этот пакет проекту команд или запросов.

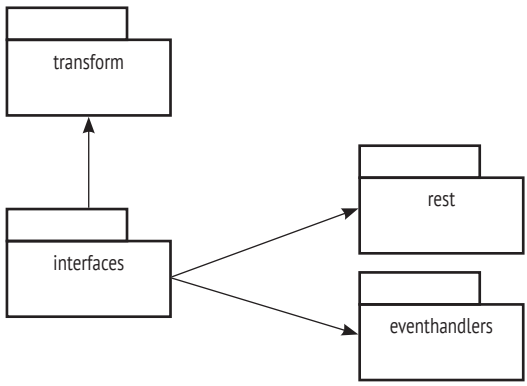

Рис. 6.24 ❖ Структура пакета interfaces

Пакет application

Следует напомнить, что сервисы приложения функционируют как внешний компонент модели предметной области (домена) ограниченного контекста. В дополнение к функциям внешнего («фасадного») уровня в рамках шаблона CQRS/ES сервисы приложения также отвечают за делегирование команд и запросов с обращением (вызовом) к модели регулирования и координации Axon Dispatch Model.

Таким образом, сервисы приложения выполняют следующие функции:

- участие в регулировании и распределении команд, запросов и в сагах;
- предоставление централизованных компонентов (например, ведение журналов, обеспечение безопасности, метрики) для модели предметной области (домена) более низкого уровня;
- выполнение вызовов, направленных в другие ограниченные контексты.

Структура пакета application показана на рис. 6.25.

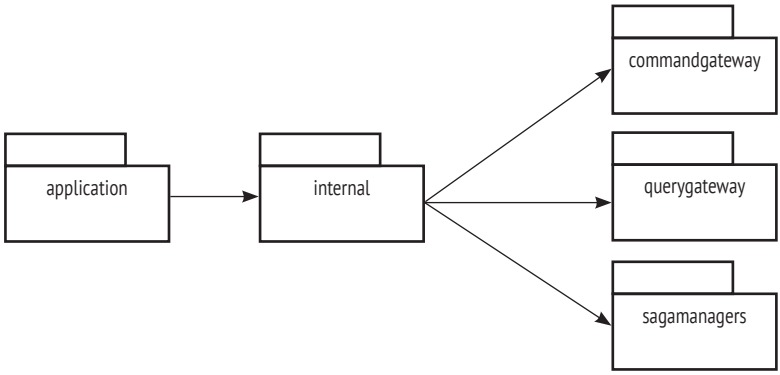

Рис. 6.25 ❖ Структура пакета application

Пакет domain

Пакет domain содержит модель предметной области (домена) ограниченного контекста.

Модель предметной области (домена) состоит из следующих объектов:

- агрегатов;
- проекций агрегатов (модель чтения);
- команд;
- запросов;
- событий;
- обработчиков запросов.

Структура пакета domain показана на рис. 6.26.

Пакет infrastructure

Пакет infrastructure предназначен для достижения следующих двух основных целей:

- создания компонентов инфраструктуры, требуемых моделью предметной области (домена) для обмена данными с любыми внешними репозиториями, например, область запросов ограниченного контекста обменивается данными с моделью чтения репозитория, например базы данных MySQL или хранилища документов MongoDB;
- создания любого типа конфигурации, специализированной для Axon, например для быстрого тестирования может потребоваться использование встроенного хранилища событий Embedded Event Store вместо хранилища событий Axon Server. Такая конфигурация должна быть размещена в классах пакета infrastructure.

Структура пакета infrastructure показана на рис. 6.27.

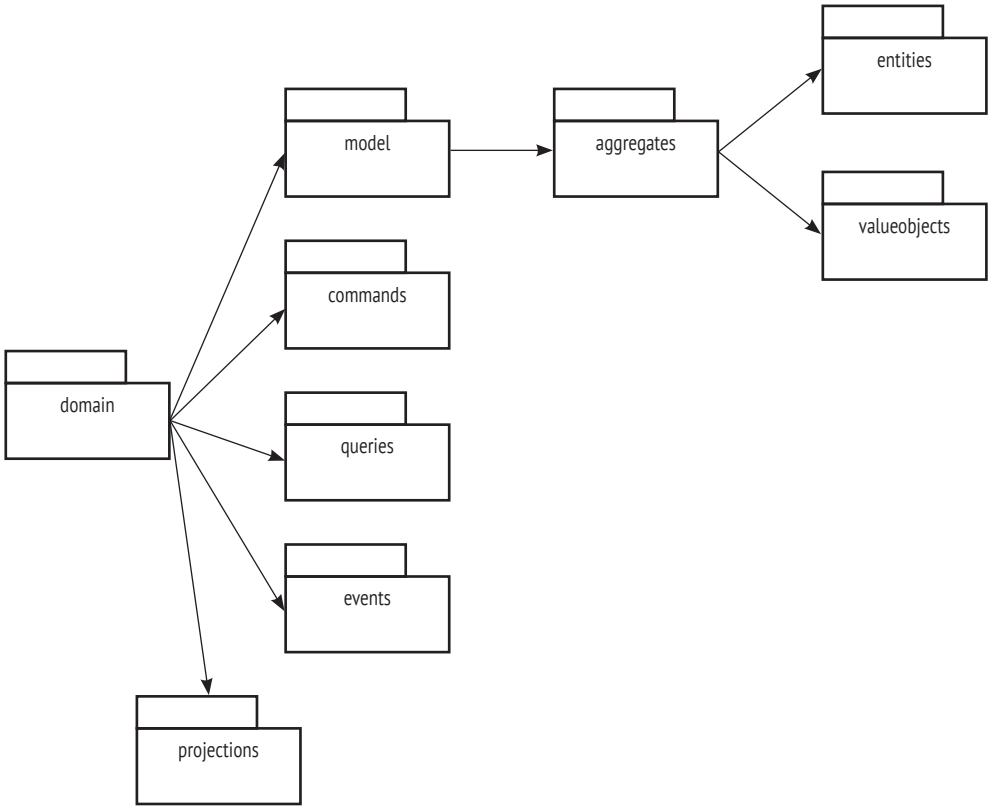

Рис. 6.26 ❖ Структура пакета domain для модели предметной области

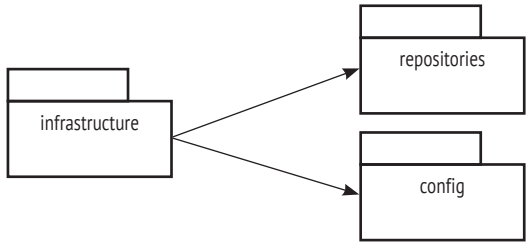

Рис. 6.27 ❖ Структура пакета infrastructure, содержащего компоненты инфраструктуры

Для примера ограниченного контекста Booking (заказ груза) полная структура пакета показана на рис. 6.28.

Результатом создания приложения для ограниченного контекста Booking является JAR-файл Spring Boot с именем *bookingms-1.0.jar*. Для запуска приложения ограниченного контекста Booking сначала необходимо активизировать Axon Server. Затем запускается ограниченный контекст Booking как обычный Spring Boot JAR-файл. Команда запуска показана в листинге 6.3.

Листинг 6.3 ❖ Команда запуска ограниченного контекста Booking как приложения Spring Boot

```
java -jar bookingqueryms-1.0.jar
```

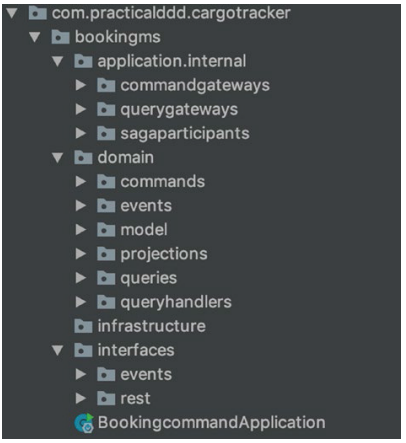

Рис. 6.28 ❖ Структура пакета для ограниченного контекста Booking

Зависимость `axon-spring-boot` для этого приложения автоматически выполняет поиск работающего сервера Axon Server и автоматически устанавливает соединение с ним. На рис. 6.29 показана панель управления Axon, в которой графически отображено установленное соединение микросервиса Booking с работающим сервером Axon Server.

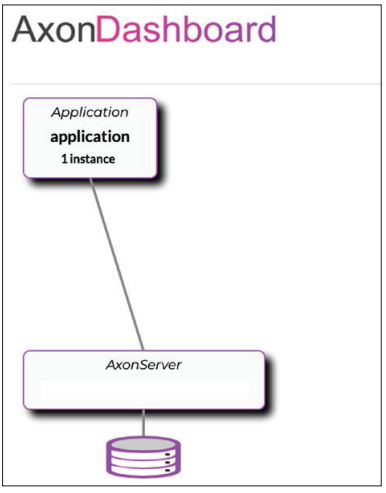

Рис. 6.29 ❖ Микросервис Booking соединен с Axon Server и готов к обработке команд и запросов

На этом завершается рассмотрение реализации ограниченных контекстов в приложении Cargo Tracker на основе микросервисов и шаблонов CQRS/ES с использованием рабочей среды Axon Framework. Каждый ограниченный контекст реализуется как приложение Axon Spring Boot. Ограниченные контексты аккуратно распределяются по модулям в структуре пакета с четким разделением компонентов.

В следующих двух разделах этой главы будет рассматриваться реализация артефактов предметно-ориентированного проектирования на основе рабочей среды Axon в приложении Cargo Tracker – модель предметной области (домена) и сервисы модели предметной области (домена). Общая схема артефактов предметно-ориентированного проектирования показана на рис. 6.30.

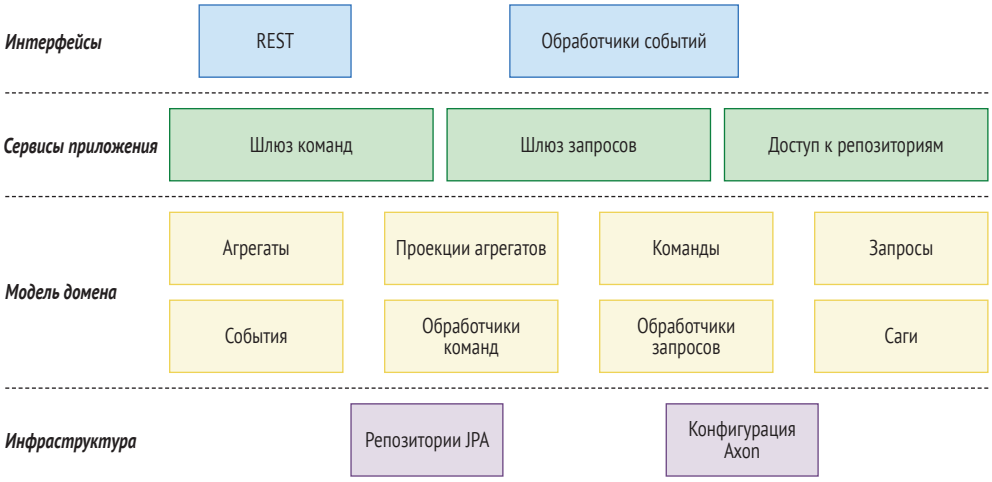

Рис. 6.30 ❖ Артефакты предметно-ориентированного проектирования на основе рабочей среды Axon

Реализация модели предметной области с использованием Axon

Модель предметной области (домена) является главным компонентом каждого ограниченного контекста, представляющего основные бизнес-функции. Рассматриваемая здесь реализация модели предметной области (домена) с использованием рабочей среды Axon абсолютно отличается от предыдущих реализаций, с учетом того что рабочая среда Axon весьма строго соблюдает принципы предметно-ориентированного проектирования и CQRS/ES.

В этом разделе будет рассматриваться реализация следующего набора артефактов для модели предметной области (домена):

- агрегаты/команды;
- проекции агрегатов (модель чтения)/запросы;
- события;
- саги.

Агрегаты

Агрегаты являются основным компонентом модели предметной области (домена) в ограниченном контексте. Поскольку в рассматриваемой здесь реализации применяется шаблон CQRS, создаются два ограниченных контекста для каждого поддомена: один для области команд, другой для области запросов. В первую очередь потребуются агрегаты для ограниченного контекста области

команд, затем будут реализованы проекции агрегатов для ограниченного контекста области запросов.

На рис. 6.31 показаны агрегаты для каждого ограниченного контекста области команд.

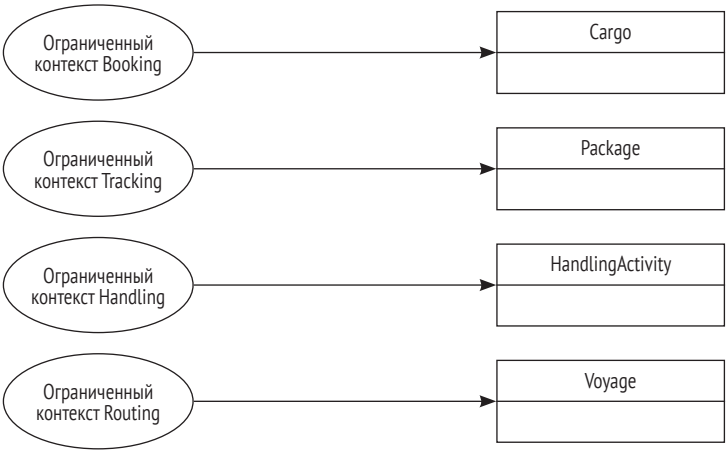

Рис. 6.31 ❖ Агрегаты для каждого ограниченного контекста области команд

Реализация класса агрегата охватывает следующие аспекты:

- реализацияю класса агрегата;
- состояние;
- обработку команд;
- публикацию событий;
- сопровождение состояния.

Рабочая среда Axon предоставляет поддержку объектов первого класса для создания классов агрегатов с использованием стандартных аннотаций. Первый шаг реализации агрегата – создание обычного простого старого объекта Java (POJO) и его пометка предоставляемой рабочей средой Axon аннотацией `@Aggregate`. Такая аннотация сообщает рабочей среде о том, что это класс агрегата в ограниченном контексте.

Как и ранее, здесь будет подробно рассматриваться реализация агрегата Cargo, который является агрегатом для примера ограниченного контекста Booking (область команд).

В листинге 6.4 показан первый шаг реализации агрегата Cargo.

Листинг 6.4 ❖ Агрегат Cargo с использованием аннотации Axon

```
package com.practicalddd.cargotracker.bookingms.domain.model;
import org.axonframework.spring.stereotype.Aggregate;
@Aggregate // Axon предоставляет аннотацию для пометки Cargo как агрегата.
public class Cargo {
}
```

Следующий шаг – обеспечение уникальности агрегата, т. е. создание ключа для идентификации каждого экземпляра агрегата. Наличие идентификато-

ра агрегата обязательно, так как рабочая среда использует его, для того чтобы определить, какой экземпляр агрегата необходимо выбрать для обработки конкретной команды. Axon предоставляет стандартную аннотацию `@AggregateIdentifier` для определения конкретного поля агрегата как его идентификатора.

Продолжим рассмотрение примера реализации агрегата Cargo: идентификатор заказа груза BookingId – это идентификатор агрегата Booking, как показано в листинге 6.5.

Листинг 6.5 ❖ Реализация идентификатора агрегата с использованием аннотации Axon

```
package com.practicalddd.cargotracker.bookingms.domain.model;
import org.axonframework.spring.stereotype.Aggregate;
import org.axonframework.modelling.command.AggregateIdentifier;
@Aggregate     // Axon предоставляет аннотацию для пометки Cargo как агрегата.
public class Cargo {
  @AggregateIdentifier  // Axon предоставляет аннотацию для пометки BookingID
                        // как идентификатора агрегата.
  private String bookingId;
}
```

Завершающим шагом этой реализации является создание конструктора без аргументов. Это требование рабочей среды в основном во время операций обновления агрегата. Axon будет использовать конструктор без аргументов для создания пустого экземпляра агрегата. Затем воспроизводятся все предыдущие события, произошедшие в этом экземпляре агрегата, для достижения самого последнего текущего состояния. Более подробно эта тема будет обсуждаться позже в разделе о сопровождении состояния. Сейчас достаточно просто разместить конструктор без аргументов в реализации агрегата.

В листинге 6.6 показано добавление конструктора без аргументов в реализацию агрегата.

Листинг 6.6 ❖ Конструктор агрегата Cargo

```
package com.practicalddd.cargotracker.bookingms.domain.model;
import org.axonframework.spring.stereotype.Aggregate;
import org.axonframework.modelling.command.AggregateIdentifier;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.lang.invoke.MethodHandles;
@Aggregate // Axon предоставляет аннотацию для пометки Cargo как агрегата.
public class Cargo {
  private final static Logger logger =
               LoggerFactory.getLogger( MethodHandles.lookup().lookupClass() );
  @AggregateIdentifier  // Axon предоставляет аннотацию для пометки BookingID
                        // как идентификатора агрегата.
  private String bookingId;

  protected Cargo() { // Пустой конструктор без аргументов.
    logger.info("Empty Cargo created.");
  }
}
```

Схема реализации класса агрегата показана на рис. 6.32. Следующий шаг – добавление состояния.

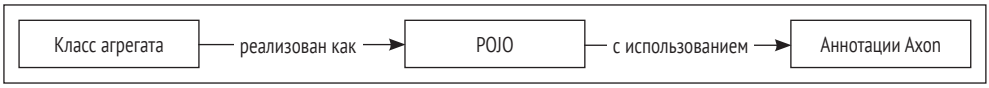

Рис. 6.32 ❖ Реализация класса агрегата

Состояние

Полноценные и неполноценные агрегаты обсуждались в главе 3 при рассмотрении реализации на основе платформы Jakarta EE. Методика предметно-ориентированного проектирования рекомендует использовать агрегаты полноценной предметной области (домена), которые содержат состояние ограниченного контекста, используя для этого бизнес-концепции.

Рассмотрим вариант класса корневого агрегата Cargo в ограниченном контексте Booking (заказ груза). Суть предметно-ориентированного проектирования заключается в захвате состояния агрегата как атрибутов, отображаемых с использованием бизнес-терминологии, а не технических терминов.

После преобразования состояния в бизнес-концепции агрегат Cargo содержит следующие атрибуты:

- исходную локацию груза;
- количество груза;
- спецификацию маршрута (исходную локацию, целевую локацию (пункт назначения), предельный срок доставки в целевую локацию);
- план маршрута доставки, который назначается грузу на основе спецификации маршрута. План маршрута (Itinerary) состоит из нескольких этапов (Legs), по которым может быть направлен груз, чтобы достичь целевой локации.

На рис. 6.33 показана UML-диаграмма класса для агрегата Cargo с соответствующими связями (ассоциациями).

Теперь необходимо включить перечисленные выше атрибуты в агрегат Cargo. Эти атрибуты реализуются как обычные простые старые объекты Java (POJO) с установлением строгого ассоциативного отношения с агрегатом.

В листинге 6.7 показан основной код объекта агрегата Cargo.

Листинг 6.7 ❖ Реализация агрегата Cargo

```
package com.practicalddd.cargotracker.bookingms.domain.model;
import java.lang.invoke.MethodHandles;
import org.axonframework.modelling.command.AggregateIdentifier;
import org.axonframework.spring.stereotype.Aggregate;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
@Aggregate
public class Cargo {
  private final static Logger logger =
                       LoggerFactory.getLogger( MethodHandles.lookup().lookupClass() );
  @AggregateIdentifier
```

```
  private String bookingId;    // Идентификатор агрегата.
  private BookingAmount bookingAmount;   // Количество заказанного груза.
  private Location origin;   // Исходная локация груза.
  private RouteSpecification routeSpecification;   // Спецификация маршрута груза.
  private Itinerary itinerary;   // План маршрута, назначенный для груза.
  protected Cargo() { logger.info( "Empty Cargo created" ); }
}
```

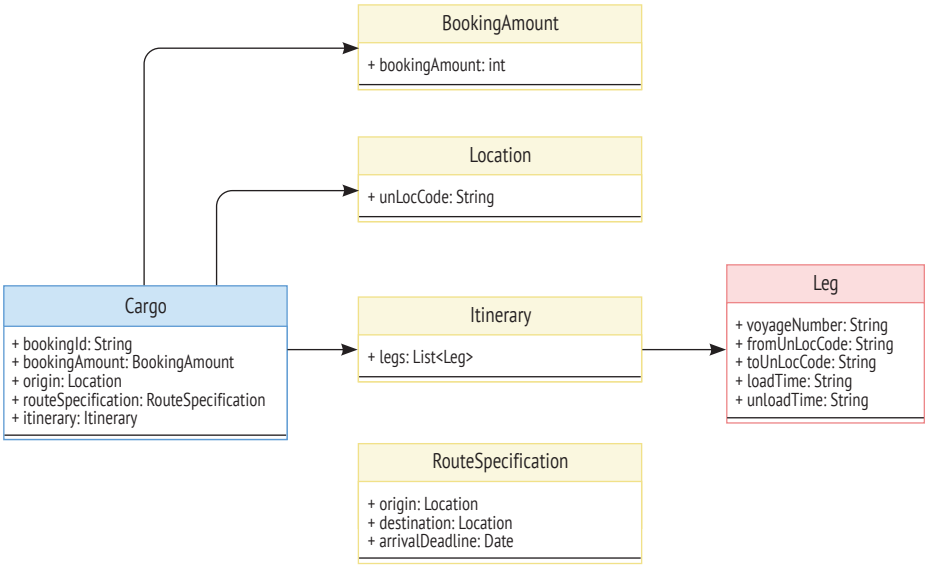

Рис. 6.33 ❖ Диаграмма класса агрегата Cargo

В листинге 6.8 показана реализация бизнес-объекта BookingAmount.

```
package com.practicalddd.cargotracker.bookingms.domain.model;

/**
 * Реализация объекта BookingAmount агрегата Cargo.
 */
public class BookingAmount {
  private int bookingAmount;
  public BookingAmount() {}
  public BookingAmount( int bookingAmount ) {
    this.bookingAmount = bookingAmount;
  }
}
```

В листинге 6.9 показана реализация бизнес-объекта Location.

Листинг 6.9 ❖ Бизнес-объект Location

```
package com.practicalddd.cargotracker.bookingms.domain.model;
/**
 * Класс Location, представленный уникальным 5-значным кодом UN Location.
 */
```

```
public class Location {
  private String unLocCode; // Код локации ООН (UN location.
  public Location( String unLocCode ) { this.unLocCode = unLocCode; }
  public void setUnLocCode( String unLocCode ) { this.unLocCode = unLocCode; }
  public String getUnLocCode() { return this.unLocCode; }
}
```

В листинге 6.10 показана реализация бизнес-объекта RouteSpecification.

Листинг 6.10 ❖ Бизнес-объект RouteSpecification

```
package com.practicalddd.cargotracker.bookingms.domain.model;
import java.util.Date;
/**
 * Спецификация маршрута груза – исходный пункт, пункт назначения и предельный срок
доставки.
 */
public class RouteSpecification {
  private Location origin;
  private Location destination;
  private Date arrivalDeadline;
  public RouteSpecification( Location origin, Location destination, Date arrivalDeadline )
{
    this.setOrigin( origin );
    this.setDestination( destination );
    this.setArrivalDeadline( (Date) arrivalDeadline.clone() );
  }
  public Location getOrigin() { return origin; }
  public void setOrigin( Location origin ) { this.origin = origin; }
  public Location getDestination() { return destination; }
  public void setDestination( Location destination ) { this.destination = destination; }
  public Date getArrivalDeadline() { return arrivalDeadline; }
  public void setArrivalDeadline( Date arrivalDeadline ) {
    this.arrivalDeadline = arrivalDeadline;
  }
}
```

В листинге 6.11 показана реализация бизнес-объекта Itinerary.

Листинг 6.11 ❖ Бизнес-объект Itinerary

```
package com.practicalddd.cargotracker.bookingms.domain.model;
import java.util.Collections;
import java.util.List;
/**
 * План маршрута доставки груза. Состоит из этапов Legs, которые будет проходить груз,
 * как частей полного пути доставки.
 */
public class Itinerary {
  private List<Leg> legs = Collections.emptyList();
  public Itinerary() {}
  public Itinerary( List<Leg> legs ) {
    this.legs = legs;
  }
```

```
  public List<Leg> getLegs() {
    return Collections.unmodifiableList(legs);
  }
}
```

В листинге 6.12 показана реализация бизнес-объекта Leg.

Листинг 6.12 ❖ Бизнес-объект Leg

```
package com.practicalddd.cargotracker.bookingms.domain.model;
/**
 * Этап (Leg) в плане маршрута, который в текущий момент назначен для груза Cargo.
 */
public class Leg {
  private  String voyageNumber;
  private  String fromUnLocode;
  private  String toUnLocode;
  private  String loadTime;
  private  String unloadTime;
  public Leg(
            String voyageNumber,
            String fromUnLocode,
            String toUnLocode,
            String loadTime,
            String unloadTime ) {
    this.voyageNumber = voyageNumber;
    this.fromUnLocode = fromUnLocode;
    this.toUnLocode = toUnLocode;
    this.loadTime = loadTime;
    this.unloadTime = unloadTime;
  }

  public String getVoyageNumber() {
    return voyageNumber;
  }

  public String getFromUnLocode() {
    return fromUnLocode;
  }

  public String getToUnLocode() {
    return toUnLocode;
  }

  public String getLoadTime() { return loadTime; }

  public String getUnloadTime() {
    return unloadTime;
  }

  public void setVoyageNumber( String voyageNumber ) {
    this.voyageNumber =  voyageNumber;
  }

  public void setFromUnLocode( String fromUnLocode ) {
    this.fromUnLocode = fromUnLocode;
  }
```

```
  public void setToUnLocode(String toUnLocode) {
    this.toUnLocode =  toUnLocode;
  }

  public void setLoadTime( String loadTime ) { this.loadTime = loadTime; }

  public void setUnloadTime( String unloadTime ) { this.unloadTime = unloadTime; }

  @Override
  public String toString() {
    return "Leg{" + "voyageNumber=" + voyageNumber + ", from="
                  + fromUnLocode + ", to=" + toUnLocode + ", loadTime=" + loadTime
                  + ", unloadTime=" + unloadTime + '}';
  }
}
```

Схема реализации состояния агрегата показана на рис. 6.34. Следующий шаг – реализация обработки команд.

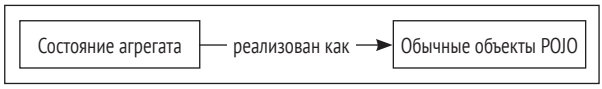

Рис. 6.34 ❖ Реализация состояния агрегата

Обработка команд

Команды передают ограниченному контексту инструкции для изменения его состояния, чаще всего для изменения состояния агрегата (или любых других идентифицируемых сущностей). Реализация обработки команд подразумевает выполнение следующих шагов:

- идентификации/реализации команд;
- идентификации/реализации обработчиков команд для обработки команд.

Идентификация команд

Идентификация команд охватывает любую операцию, которая влияет на состояние агрегата. Например, ограниченный контекст команд заказа груза содержит следующие операции или команды:

- заказ груза;
- назначение маршрута доставки груза;
- изменение целевой локации (пункта назначения) доставки груза.

Все три перечисленные команды в конечном итоге изменяют состояние агрегата Cargo в соответствующем ограниченном контексте.

Реализация команд

Реализация идентифицированных команд в рабочей среде Axon выполняется с использованием обычных старых объектов Java (POJO). Единственное требование для Axon-объекта команды: при обработке команды рабочая среда Axon обязательно должна знать, в каком экземпляре агрегата необходимо выполнить обработку этой конкретной команды. Это требование удовлетворяется с помощью аннотации Axon `@TargetAggregateIdentifier`. По наименованию ан-

нотации можно понять, что при обработке команды рабочая среда Axon узнает целевой экземпляр агрегата, в котором должна обрабатываться эта конкретная команда.

Рассмотрим пример. В листинге 6.13 показана реализация класса `BookCargoCommand` для команды заказа груза.

Листинг 6.13 ❖ Реализация класса BookCargoCommand

```
package com.practicalddd.cargotracker.bookingms.domain.commands;
import org.axonframework.modelling.command.TargetAggregateIdentifier;
import java.util.Date;

/**
 * Реализация класса для команды заказа груза BookCargoCommand.
 */
public class BookCargoCommand {
  @TargetAggregateIdentifier   // Идентификатор для определения экземпляра агрегата,
                               // который должен обрабатывать эту команду.
  private String bookingId;   // BookingId - уникальный ключ для идентификации агрегата.
  private int bookingAmount;
  private String originLocation;
  private String destLocation;
  private Date destArrivalDeadline;
  public BookCargoCommand( String bookingId, int bookingAmount, String originLocation,
                           String destLocation, Date destArrivalDeadline ) {
    this.bookingId = bookingId;
    this.bookingAmount = bookingAmount;
    this.originLocation = originLocation;
    this.destLocation = destLocation;
    this.destArrivalDeadline = destArrivalDeadline;
  }

  public void setBookingId( String bookingId ) { this.bookingId = bookingId; }

  public void setBookingAmount( int bookingAmount ) { this.bookingAmount = bookingAmount; }

  public String getBookingId() { return this.bookingId; }

  public int getBookingAmount() { return this.bookingAmount; }

  public String getOriginLocation() { return originLocation; }

  public void setOriginLocation( String originLocation ) {
    this.originLocation = originLocation;
  }

  public String getDestLocation() { return destLocation; }

  public void setDestLocation( String destLocation ) { this.destLocation = destLocation; }

  public Date getDestArrivalDeadline() { return destArrivalDeadline; }

  public void setDestArrivalDeadline( Date destArrivalDeadline ) {
    this.destArrivalDeadline = destArrivalDeadline;
  }
}
```

Класс `BookCargoCommand` – это обычный простой старый объект Java (POJO), который содержит все необходимые атрибуты для обработки команды заказа груза (BookingId, количество груза, исходную и конечную локации, предельный срок доставки).

Идентификатор BookingId представляет неповторяемость агрегата Cargo, т. е. собственно идентификатор самого агрегата. Для поля BookingId назначается аннотация, определяющая идентификатор целевого агрегата. Таким образом, каждый раз, когда команда передается в ограниченный контекст обработки команды заказа груза, она будет обрабатываться именно в том экземпляре агрегата, который идентифицирован по BookingId.

Идентификатор агрегата обязательно должен устанавливаться перед выполнением любой команды в рабочей среде Axon, поскольку без идентификатора Axon не узнает, какой экземпляр агрегата должен обработать эту команду.

Идентификация обработчиков команд

Для каждой команды будет предусмотрен соответствующий обработчик команды (Command Handler). Команда BookCargoCommand будет иметь соответствующий обработчик, который принимает эту команду в качестве входного параметра и обрабатывает ее. Обработчики команд обычно размещаются в подпрограммах внутри агрегатов. Но рабочая среда Axon также позволяет размещать обработчики команд вне агрегатов на уровне сервисов приложения.

Реализация обработчиков команд

Реализация обработчиков команд, как и ранее, подразумевает в агрегатах идентификацию подпрограмм, которые могут обрабатывать команды. Axon предоставляет соответствующую аннотацию `@CommandHandler`, которая определяет в агрегатах подпрограммы, идентифицированные как обработчики команд.

Рассмотрим пример обработчика команды заказа груза CargoBookingCommand. В листинге 6.14 показано использование аннотаций `@CommandHandler`, определяющих конструкторы агрегата и обычные подпрограммы (методы).

Листинг 6.14 ❖ Обработчики команд в агрегате Cargo

```
package com.practicalddd.cargotracker.bookingms.domain.model;

import java.lang.invoke.MethodHandles;
import com.practicalddd.cargotracker.bookingms.domain.commands.AssignRouteToCargoCommand;
import com.practicalddd.cargotracker.bookingms.domain.commands.BookCargoCommand;
import com.practicalddd.cargotracker.bookingms.domain.commands.ChangeDestinationCommand;
import org.axonframework.commandhandling.CommandHandler;
import org.axonframework.modelling.command.AggregateIdentifier;
import org.axonframework.spring.stereotype.Aggregate;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Aggregate
public class Cargo {
      private final static Logger logger =
                        LoggerFactory.getLogger( MethodHandles.lookup().lookupClass() );
  @AggregateIdentifier
```

```
  private String bookingId; // Идентификатор агрегата.
  private BookingAmount bookingAmount; // Количество груза.
  private Location origin; // Исходная локация груза.
  private RouteSpecification routeSpecification; // Спецификация маршрута доставки груза.
  private Itinerary itinerary; // План маршрута, назначенный для этого груза.
  protected Cargo() {
    logger.info( "Empty Cargo created." );
  }

  @CommandHandler // Обработчик команды BookCargoCommand. Первая команда, переданная в
                  // агрегат, обрабатывается в конструкторе агрегата.
  public Cargo( BookCargoCommand bookCargoCommand ) {
    // Обработка команды.
  }

  @CommandHandler // Обработчик команды назначения маршрута доставки груза.
  public void handle( AssignRouteToCargoCommand assignRouteToCargoCommand ) {
    // Обработка команды.
  }

  @CommandHandler // Обработчик команды изменения пункта назначения груза.
  public void handle( ChangeDestinationCommand changeDestinationCommand ) {
    // Обработка команды.
  }
}
```

Обычно самая первая команда, передаваемая в агрегат, определяет создание агрегата, и поэтому ее обработка размещается в конструкторе агрегата (также называемого конструктором с обработкой команд).

Обработка последующих команд размещается в обычных подпрограммах (методах) в агрегате. В рассматриваемом примере обработчики команд `RouteCargoCommand` и `ChangeCargoDestinationCommand` размещены в обычных подпрограммах (методах) агрегата Cargo.

Обработчики команд должны обеспечивать бизнес-логику и принятие решений (например, проверку даты создания обрабатываемой команды), чтобы обеспечить последующую правильную обработку и генерацию событий. Эта ответственность целиком и полностью ложится на обработчики команд: ничто иное не должно изменять состояние агрегата.

В листинге 6.15 показан пример кода из обработчика команды BookCargoCommand, где проверяется корректность количества заказанного груза.

Листинг 6.15 ❖ Бизнес-логика и принятие решений в обработчиках команд агрегата

```
@CommandHandler
public Cargo( BookCargoCommand bookCargoCommand ) {
  // Проверка корректности количества заказанного груза. Исключение генерируется,
  // если это отрицательное число.
  if( bookCargoCommand.getBookingAmount() < 0 ) {
    throw new IllegalArgumentException( "Booking Amount cannot be negative" );
  }
}
```

На рис. 6.35 показана диаграмма класса агрегата Cargo с соответствующими командами и обработчиками команд.

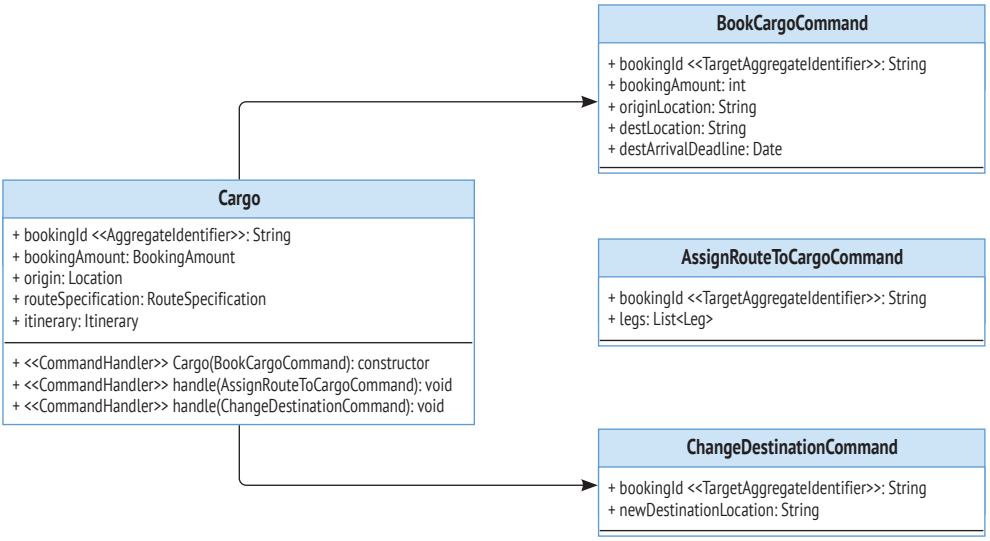

Рис. 6.35 ❖ Диаграмма класса агрегата Cargo

Реализация обработчика команд показана на рис. 6.36.

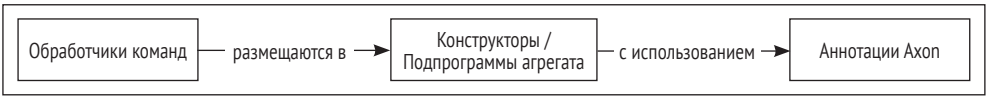

Рис. 6.36 ❖ Реализация обработчика команд

Следующий шаг – публикация событий.

Публикация событий

После обработки команды и выполнения всех операций бизнес-логики и принятия решений необходимо опубликовать событие, соответствующее обработанной команде; например, в результате обработки команды BookCargoCommand генерируется событие CargoBookedEvent, которое необходимо опубликовать. Наименования событий всегда записываются в прошедшем времени, так как они обозначают то, что уже произошло в ограниченном контексте.

При публикации событий выполняются следующие шаги:

- идентификация и реализация событий;
- реализация публикации событий.

Идентификация и реализация событий

В результате обработки каждой команды всегда генерируется некоторое событие. Команда BookCargoCommand генерирует событие CargoBookedEvent, команда AssignRouteToCargoCommand генерирует событие CargoRoutedEvent.

События передают состояние агрегата после обработки и выполнения конкретной команды, поэтому становится чрезвычайно важным обеспечение того, что события содержат все требуемые данные. Классы событий реализуются как простые старые объекты Java (POJO), которые не требуют каких-либо стандартных аннотаций.

В листинге 6.16 показан пример реализации события `CargoBookedEvent`.

Листинг 6.16 ❖ Реализация события CargoBookedEvent

```
package com.practicalddd.cargotracker.bookingms.domain.events;

import com.practicalddd.cargotracker.bookingms.domain.model.BookingAmount;
import com.practicalddd.cargotracker.bookingms.domain.model.Location;
import com.practicalddd.cargotracker.bookingms.domain.model.RouteSpecification;

/**
 * Событие, генерируемое в результате обработки команды заказа груза.
 */
public class CargoBookedEvent {
  private String bookingId;
  private BookingAmount bookingamount;
  private Location originLocation;
  private RouteSpecification routeSpecification;

  public CargoBookedEvent( String bookingId,  BookingAmount bookingAmount,
                           Location originLocation, RouteSpecification
                           routeSpecification ) {
    this.bookingId = bookingId;
    this.bookingamount = bookingAmount;
    this.originLocation = originLocation;
    this.routeSpecification = routeSpecification;
  }

  public String getBookingId() { return this.bookingId; }
  public BookingAmount getBookingAmount() { return this.bookingamount; }
  public Location getOriginLocation() { return this.originLocation; }
  public RouteSpecification getRouteSpecification() { return this.routeSpecification; }
}
```

Реализация публикации событий

Где публиковать сгенерированные события? Там же, где они были сгенерированы, т. е. в соответствующем обработчике команд. Возвращаясь к реализации в предыдущих разделах, напомним, что обработчики команд обрабатывают команды и после завершения обработки они отвечают за публикацию соответствующего события.

Обработка команд является частью жизненного цикла экземпляра агрегата. Рабочая среда Axon предоставляет класс `AggregateLifeCycle`, который помогает выполнять операции на протяжении жизненного цикла агрегата. Этот класс предоставляет статический метод (функцию) `apply()`, которая помогает опубликовать сгенерированные события.

В листинге 6.17 показан фрагмент кода из обработчика команды `BookCargoCommand`. После завершения обработки команды вызывается метод `apply()` для публикации события «груз заказан» с классом `CargoBookedEvent` как полезной нагрузкой этого события.

Листинг 6.17 ❖ Публикация события CargoBookedEvent

```
@CommandHandler
public Cargo( BookCargoCommand bookCargoCommand ) {
  logger.info( "Handling {}", bookCargoCommand );
  if( bookCargoCommand.getBookingAmount() < 0 ) {
    throw new IllegalArgumentException( "Booking Amount cannot be negative" );
  }
  // Публикация сгенерированного события с использованием метода apply.
  apply(new CargoBookedEvent( bookCargoCommand.getBookingId(),
                              new BookingAmount( bookCargoCommand.getBookingAmount() ),
                              new Location( bookCargoCommand.getOriginLocation() ),
                              new RouteSpecification(
                                    new Location( bookCargoCommand.getOriginLocation() ),
                                    new Location( bookCargoCommand.getDestLocation() ),
                                    bookCargoCommand.getDestArrivalDeadline() ) ) );
}
```

Публикация событий показана на рис. 6.37. Следующий шаг – сопровождение состояния.

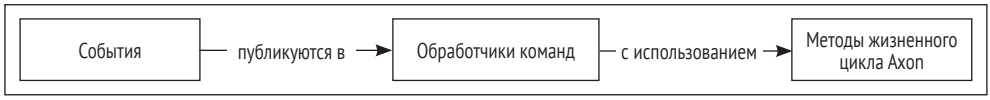

Рис. 6.37 ❖ Реализация публикации событий

Сопровождение состояния

Самой важной и чрезвычайно ответственной частью процесса шаблона Event Sourcing является понимание того, как сопровождается и используется состояние. Этот раздел содержит некоторые весьма важные концепции, относящиеся к логической целостности и непротиворечивости состояния. В отличие от теоретической литературы здесь эти концепции будут описываться на конкретных практических примерах.

Снова будет рассматриваться пример команды заказа груза в ограниченном контексте Booking. Здесь еще раз следует напомнить, что в текущий момент агрегат (Cargo) уже идентифицирован, ему назначен план маршрута доставки, обработаны команды и опубликованы события.

Для объяснения концепции сопровождения состояния (state maintenance) необходимо добавить атрибут в агрегат Cargo.

RoutingStatus – этот атрибут определяет состояние маршрута заказанного груза:

- для нового заказанного груза маршрут пока еще не назначен, так как компания, поставляющая этот груз, принимает решение по выбору наиболее оптимального маршрута (состояние маршрута – `NOT_ROUTED`);
- компания, поставляющая груз, принимает решение по выбору наиболее оптимального маршрута и назначает этот маршрут для заказанного груза (состояние маршрута – `ROUTED`).

В листинге 6.18 показана реализация состояния маршрута как перечисления (Enum).

Листинг 6.18 ❖ Реализация состояния маршрута как перечисления

```
package com.practicalddd.cargotracker.bookingms.domain.model;

/**
 * Класс перечисления для состояния маршрута заказанного груза.
 */
public enum RoutingStatus {
  NOT_ROUTED, ROUTED, MISROUTED;
  public boolean sameValueAs( RoutingStatus other ) {
    return this.equals( other );
  }
}
```

Обработка событий в агрегатах

При публикации события из агрегата рабочая среда Axon в первую очередь обеспечивает доступность этого события для самого агрегата. Поскольку агрегат соответствует шаблону Event Sourcing, он полагается на события для сопровождения своего состояния. Эта концепция немного трудна для понимания с первого раза, поэтому для сравнения вспомним обычный способ извлечения и сопровождения состояния агрегата. Проще говоря, агрегат зависит от источника событий, а не от обычного источника данных (например, базы данных) для сопровождения своего состояния.

Для обработки генерируемых событий агрегат использует аннотацию `@EventSourcingHandler`, предоставляемую рабочей средой Axon. Эта аннотация определяет, что агрегат соответствует шаблону Event Sourcing и полагается на переданное ему событие для сопровождения своего состояния.

Механизм извлечения и сопровождения состояния агрегата отличается для первой команды, принятой агрегатом, по сравнению с последующими принимаемыми командами. Мы рассмотрим оба эти варианта на примерах в следующих подразделах.

Сопровождение состояния: первая команда

Когда агрегат принимает самую первую команду, рабочая среда Axon распознает ее и не создает повторно состояние, так как состояния этого конкретного агрегата пока еще не существует. Команды, размещенные в конструкторе (Command Constructors), определяют, что это самая первая команда, принятая данным агрегатом.

Рассмотрим подробнее сопровождение состояния в этом конкретном случае.

В листинге 6.19 показаны все атрибуты, представляющие состояние агрегата Cargo.

Листинг 6.19 ❖ Реализация идентификатора агрегата с использованием аннотаций Axon

```
@AggregateIdentifier
private String bookingId;   // Идентификатор агрегата.
private BookingAmount bookingAmount;   // Количество заказанного груза.
private Location origin;   // Исходная локация груза.
private RouteSpecification routeSpecification;   // Спецификация маршрута доставки груза.
private Itinerary itinerary;   // План маршрута доставки, назначенный для этого груза.
private RoutingStatus routingStatus;   // Состояние маршрута груза.
```

После публикации события `CargoBookedEvent` в обработчике команды `BookCargoCommandHandler` необходимо установить соответствующим образом атрибуты состояния. Рабочая среда Axon передает событие `CargoBookedEvent` в первую очередь в агрегат Cargo. Агрегат Cargo обрабатывает это событие для установки и сопровождения атрибутов своего состояния.

В листинге 6.20 показана обработка агрегатом Cargo события `CargoBookedEvent`, переданного с использованием аннотации `@EventSourcingHandler`, и установка соответствующих атрибутов состояния. Настоятельно рекомендуется устанавливать идентификатор агрегата (в данном случае BookingId) в самом первом событии, обрабатываемом агрегатом.

Листинг 6.20 ❖ Реализация обработчика событий по шаблону Event Sourcing

```
@EventSourcingHandler   // Аннотация, определяющая, этот агрегат соответствует шаблону
                        // Event Sourced, а также заинтересован в событии "груз заказан",
                        // генерируемым командой заказа груза BookCargoCommand.
public void on( CargoBookedEvent cargoBookedEvent ) {
  logger.info( "Applying {}", cargoBookedEvent );
  // Сопровождение состояния.
  bookingId = cargoBookedEvent.getBookingId();
  // Строгое требование по установке.
  bookingAmount = cargoBookedEvent.getBookingAmount();
  origin = cargoBookedEvent.getOriginLocation();
  routeSpecification = cargoBookedEvent.getRouteSpecification();
  routingStatus = RoutingStatus.NOT_ROUTED;
}
```

Полная реализация показана в листинге 6.21.

Листинг 6.21 ❖ Обработчик команды и обработчик события Event Sourcing в агрегате Cargo

```
package com.practicalddd.cargotracker.bookingms.domain.model;

import java.lang.invoke.MethodHandles;
import com.practicalddd.cargotracker.bookingms.domain.commands.BookCargoCommand;
```

```
import com.practicalddd.cargotracker.bookingms.domain.events.CargoBookedEvent;
import org.axonframework.commandhandling.CommandHandler;
import org.axonframework.eventsourcing.EventSourcingHandler;
import org.axonframework.modelling.command.AggregateIdentifier;
import org.axonframework.spring.stereotype.Aggregate;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import static org.axonframework.modelling.command.AggregateLifecycle.apply;

@Aggregate
public class Cargo {
  private final static Logger logger =
                    LoggerFactory.getLogger( MethodHandles.lookup().lookupClass() );

  @AggregateIdentifier
  private String bookingId;   // Идентификатор агрегата.
  private BookingAmount bookingAmount;   // Количество заказанного груза.
  private Location origin;   // Исходная локация груза.
  private RouteSpecification routeSpecification;
          // Спецификация маршрута доставки груза.
  private Itinerary itinerary;   // План маршрута, назначенный для груза.
  private RoutingStatus routingStatus;  // Состояние маршрута доставки груза.
  protected Cargo() { logger.info( "Empty Cargo created." ); }

  @CommandHandler   // Первая команда, переданная в агрегат.
  public Cargo( BookCargoCommand bookCargoCommand ) {
    logger.info( "Handling {}", bookCargoCommand );
    if( bookCargoCommand.getBookingAmount() < 0 ) {
      throw new IllegalArgumentException( "Booking Amount cannot be negative" );
    }

    apply( new CargoBookedEvent( bookCargoCommand.getBookingId(),
                            new BookingAmount( bookCargoCommand.getBookingAmount() ),
                            new Location( bookCargoCommand.getOriginLocation() ),
                            new RouteSpecification(
                                  new Location( bookCargoCommand.getOriginLocation() ),
                                  new Location( bookCargoCommand.getDestLocation() ),
                                      bookCargoCommand.getDestArrivalDeadline() ) ) );
  }

  @EventSourcingHandler // Обработчик события для команды BookCargoCommand.
                        // Также устанавливает различные атрибуты состояния.
  public void on( CargoBookedEvent cargoBookedEvent ) {
    logger.info( "Applying {}", cargoBookedEvent );
    // Сопровождаемое состояние.
    bookingId = cargoBookedEvent.getBookingId();
    bookingAmount = cargoBookedEvent.getBookingAmount();
    origin = cargoBookedEvent.getOriginLocation();
    routeSpecification = cargoBookedEvent.getRouteSpecification();
    routingStatus = RoutingStatus.NOT_ROUTED;
  }
}
```

Графическое представление этого потока событий показано на рис. 6.38.

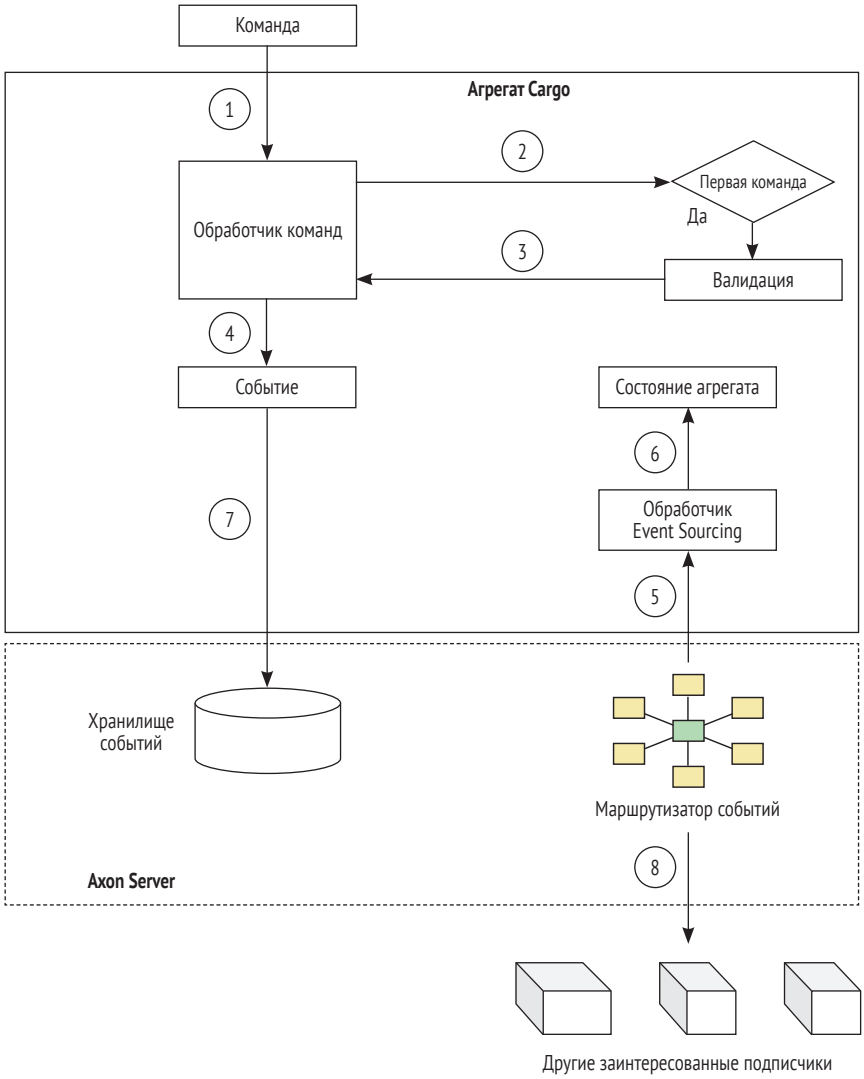

Рис. 6.38 ❖ Сопровождение состояния – самая первая команда

1. Команда направляется в соответствующий обработчик команды Cargo.
2–3. Рабочая среда Axon выполняет проверку: если это первая команда в агрегате, то после обработки управление возвращается в обработчик команды для проверки бизнес-логики.
4. Обработчик команды генерирует событие.
5. Маршрутизатор событий Axon в первую очередь обеспечивает доступность этого события для самого агрегата с помощью обработчика шаблона Event Sourcing.
6. Обработчик шаблона Event Sourcing обновляет состояние агрегата.
7. Событие сохраняется (обеспечивается его персистентность) в хранилище событий Axon.

8. Маршрутизатор событий Axon обеспечивает доступность этого события для других заинтересованных подписчиков.

К настоящему моменту обработана самая первая команда (BookCargo), опубликовано соответствующее событие (CargoBooked) и установлено состояние агрегата (Cargo).

Теперь рассмотрим, как это состояние извлекается, используется и сопровождается в последующих командах.

Сопровождение состояния: последующие команды

Когда агрегат принимает другую команду, необходимо ее обработать с учетом текущего состояния агрегата. По существу, это означает, что в начале обработки команды рабочая среда Axon обеспечивает доступность текущего состояния агрегата для обработчика команд, чтобы выполнить любые проверки бизнес-логики. Рабочая среда Axon обеспечивает такую доступность посредством загрузки пустого экземпляра агрегата, извлечения всех событий из хранилища событий и воспроизведения всех событий, которые произошли в этом конкретном экземпляре агрегата до текущего момента.

Рассмотрим подробнее это краткое описание на примере двух дополнительных команд, которые агрегат Cargo должен обработать помимо команды заказа груза, т. е. команды назначения маршрута для заказанного груза и изменения целевой локации (пункта назначения) груза.

В листинге 6.22 показан обработчик для этих двух команд.

Листинг 6.22 ❖ Реализация обработчика команд в агрегате Cargo

```
/**
 * Обработчик команды назначения маршрута для груза.
 * @param assignRouteToCargoCommand
 */
@CommandHandler
public Cargo( AssignRouteToCargoCommand assignRouteToCargoCommand ) {
  if( routingStatus.equals( RoutingStatus.ROUTED ) ) {
    throw new IllegalArgumentException( "Cargo already routed" );
  }

  apply( new CargoRoutedEvent( assignRouteToCargoCommand.getBookingId(),
                               new Itinerary( assignRouteToCargoCommand.getLegs() ) ) );
}

/**
* Cargo Handler for changing the Destination of a Cargo
* @param changeDestinationCommand
*/
@CommandHandler
public Cargo( ChangeDestinationCommand changeDestinationCommand ) {
  if( routingStatus.equals( RoutingStatus.ROUTED ) ) {
    throw new IllegalArgumentException( "Cannot change destination of a Routed Cargo" );
  }
  apply( new CargoDestinationChangedEvent( changeDestinationCommand.getBookingId(),
         new RouteSpecification( origin,
             new Location( changeDestinationCommand.getNewDestinationLocation() ),
             routeSpecification.getArrivalDeadline() ) ) );
}
```

Сначала рассмотрим обработчик команды `AssignRouteToCargoCommand`, который отвечает за обработку команды назначения маршрута доставки для заказанного груза. Этот обработчик проверяет, не назначен ли уже маршрут для заказанного груза. Для этого проверяется текущее состояние маршрута, которое представлено атрибутом `routingStatus` агрегата Cargo. Рабочая среда Axon отвечает за предоставление самого последнего установленного значения атрибута `routingStatus` в обработчик `AssignRouteToCargoCommandHandler`, т. е. за предоставление самого последнего состояния агрегата Cargo обработчику команды.

Рассмотрим, как выполняет этот процесс рабочая среда Axon по шагам.

- Рабочая среда Axon определяет, что это не первая команда в агрегате. По результату проверки Axon загружает пустой экземпляр агрегата, вызывая защищенный конструктор этого агрегата.
  В листинге 6.23 показан защищенный (protected) конструктор в агрегате Cargo.

  Листинг 6.23 ❖ Защищенный конструктор агрегата Cargo в соответствии с требованием Axon

```
protected Cargo() {
  logger.info( "Empty Cargo created." );
}
```

- Затем рабочая среда Axon запрашивает у источника событий все события, произошедшие до текущего момента в этом экземпляре агрегата, на основе идентификатора целевого агрегата, который передается в команде.
  На рис. 6.39 показана общая схема процесса, который выполняет рабочая среда Axon для получения текущего состояния агрегата.

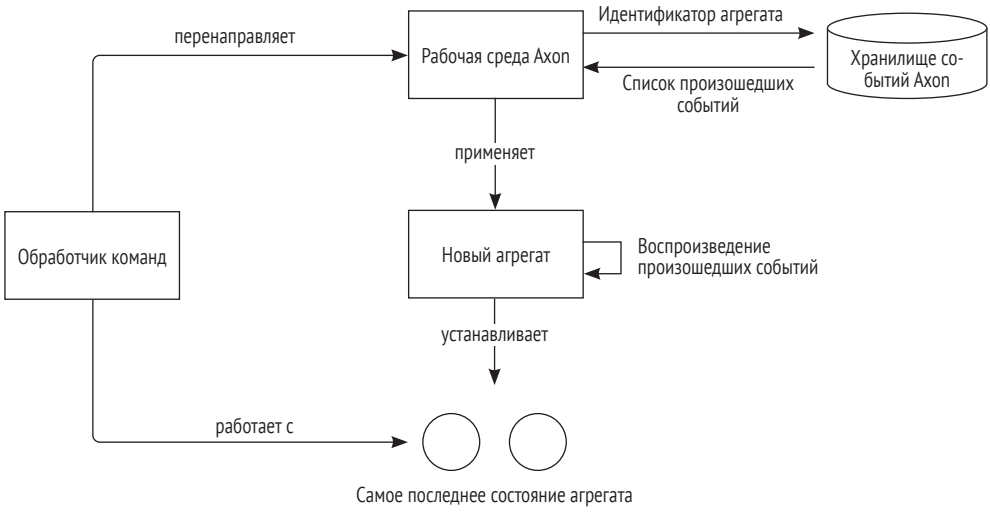

Рис. 6.39 ❖ Процесс извлечения и сопровождения состояния в рабочей среде Axon

На рис. 6.40 показан процесс, выполняемый рабочей средой Axon для получения текущего состояния агрегата Cargo после обработки команды назначения маршрута груза соответствующим обработчиком.

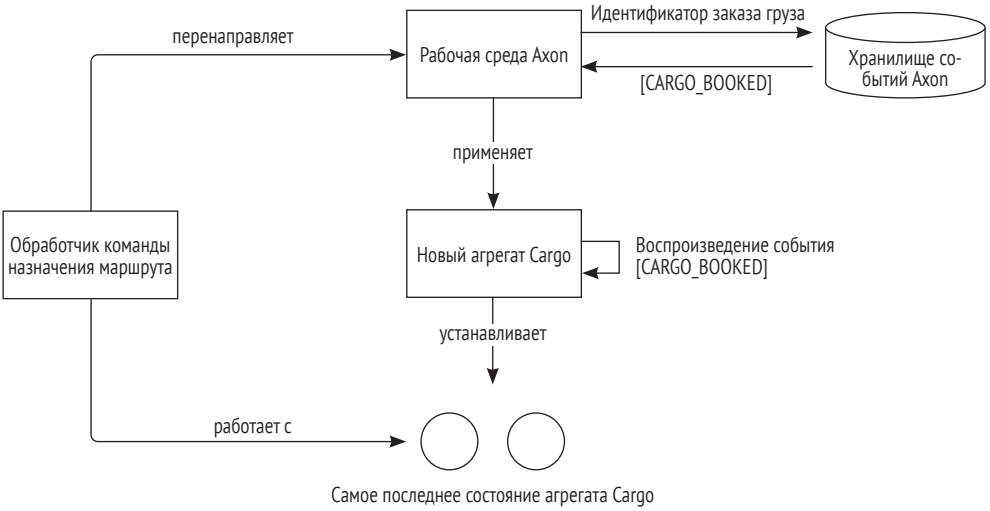

Рис. 6.40 ❖ Процесс извлечения и сопровождения состояния – команда назначения маршрута

Как показано на рис. 6.40, после приема команды рабочая среда Axon использует идентификатор агрегата (BookingId) для загрузки всех событий, которые к текущему моменту уже произошли в этом конкретном экземпляре агрегата; в рассматриваемом здесь примере это событие `CARGO_BOOKED`. Axon создает новый экземпляр агрегата с этим идентификатором и воспроизводит все ранее произошедшие события в этом экземпляре агрегата.

Воспроизведение событий, по существу, означает вызов в соответствующем агрегате отдельных методов-обработчиков Event Sourcing, которые устанавливают конкретные атрибуты состояния.

Снова обратимся к методу-обработчику Event Sourcing для события «груз заказан». См. листинг 6.24.

Листинг 6.24 ❖ Воспроизведение события в обработчике EventSourcingHandler

```
@EventSourcingHandler // Обработчик Event Sourcing для события "груз заказан".
public void on( CargoBookedEvent cargoBookedEvent ) {
  logger.info( "Applying {}", cargoBookedEvent );
  // Сопровождаемое состояние.
  bookingId = cargoBookedEvent.getBookingId();
  bookingAmount = cargoBookedEvent.getBookingAmount();
  origin = cargoBookedEvent.getOriginLocation();
  routeSpecification = cargoBookedEvent.getRouteSpecification();
  routingStatus = RoutingStatus.NOT_ROUTED;
}
```

Здесь состояние агрегата устанавливается, включая атрибут `routingStatus`. Когда это событие воспроизводится во время создания состояния агрегата, для

этого атрибута устанавливается значение NOT_ROUTED. Этот атрибут становится доступным как часть всего состояния агрегата для обработчика команды AssignRouteToCargoCommand. Так как его последним значением является NOT_ROUTED, все проверки обработчика проходят успешно, и продолжается обработка плана маршрута, назначенного для доставки этого груза, после чего новым значением атрибута routingStatus становится ROUTED.

Рассмотрим следующую передаваемую команду – ChangeDestinationCommand. В листинге 6.25 показан обработчик этой команды. Команда ChangeDestinationCommand предназначена для изменения целевой локации (пункта назначения доставки) груза. В рассматриваемом ниже обработчике команды реализуется проверка бизнес-условия: если для груза уже назначен маршрут, то нельзя разрешать изменение его целевой локации. И в этом случае проверяется атрибут агрегата routingStatus, который должен быть доступным для обработчика команды ChangeDestinationCommand.

На рис. 6.41 показан поток выполнения для команды изменения целевой локации (пункта назначения доставки) груза.

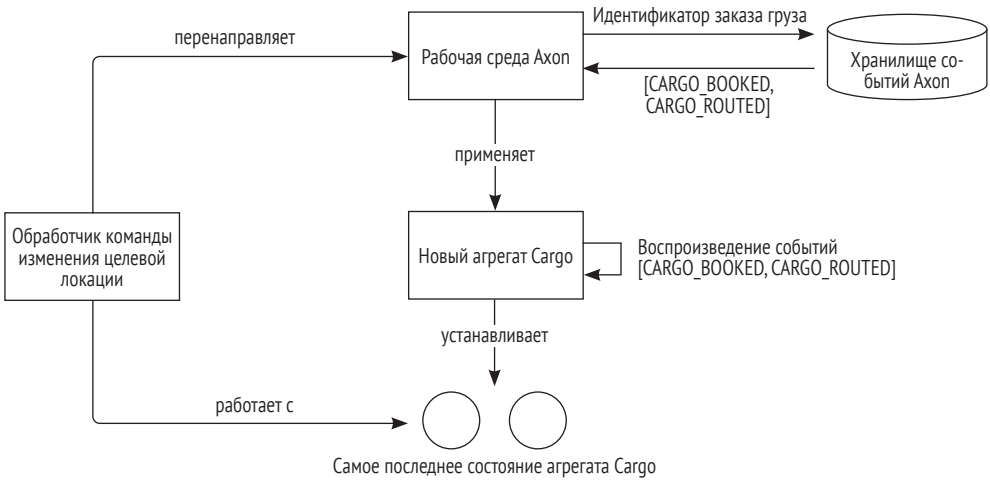

Рис. 6.41 ❖ Процесс извлечения и сопровождения состояния в рабочей среде Axon – команда изменения целевой локации после выполнения команды назначения маршрута

В рассматриваемом здесь случае рабочая среда Axon извлекает из хранилища событий два события – [CARGO_BOOKED и CARGO_ROUTED] – и воспроизводит их. И в этом случае воспроизведение событий, по существу, означает вызов в соответствующем агрегате отдельных методов-обработчиков Event Sourcing, которые устанавливают конкретные атрибуты состояния.

Снова обратимся к методу-обработчику Event Sourcing для событий «груз заказан» и «маршрут назначен». См. листинг 6.25.

```
@EventSourcingHandler   // Обработчик Event Sourcing для события "груз заказан".
public void on(CargoBookedEvent cargoBookedEvent) {
  logger.info( "Applying {}", cargoBookedEvent );
  // Сопровождаемое состояние.
  bookingId = cargoBookedEvent.getBookingId();
```

```
  bookingAmount = cargoBookedEvent.getBookingAmount();
  origin = cargoBookedEvent.getOriginLocation();
  routeSpecification = cargoBookedEvent.getRouteSpecification();
  routingStatus = RoutingStatus.NOT_ROUTED;
  // transportStatus =
  // Сохранен код оригинала – за разъяснением обращайтесь к автору книги.
}

@EventSourcingHandler // Обработчик Event Sourcing для события "маршрут назначен".
public void on( CargoRoutedEvent cargoRoutedEvent ) {
  itinerary = cargoRoutedEvent.getItinerary();
  routingStatus = RoutingStatus.ROUTED;
}
```

В настоящий момент все внимание сосредоточено на атрибуте агрегата `routingStatus`. В конце воспроизведения события (`CARGO_BOOKED`) для этого атрибута устанавливается значение `NOT_ROUTED`. В конце воспроизведения второго события (`CARGO_ROUTED`) для этого атрибута устанавливается значение `ROUTED`. Это самое последнее текущее значение рассматриваемого атрибута, которое передается как часть общего состояния агрегата в обработчик команды изменения целевой локации (пункта назначения доставки) груза. Поскольку этот обработчик команды проверяет назначение маршрута – для груза не должен быть назначен маршрут, – дальнейшая обработка запрещается и генерируется исключение (так как атрибут `routingStatus` имеет значение `ROUTED`).

С другой стороны, если команда изменения целевой локации (пункта назначения доставки) груза вызвана до выполнения команды назначения маршрута, то обработчик должен разрешить продолжение обработки, так как в агрегате Cargo было воспроизведено только одно событие (`CARGO_BOOKED`). На рис. 6.42 показан поток обработки при вызове команды изменения целевой локации (пункта назначения доставки) груза после выполнения команды заказа груза, но до выполнения команды назначения маршрута.

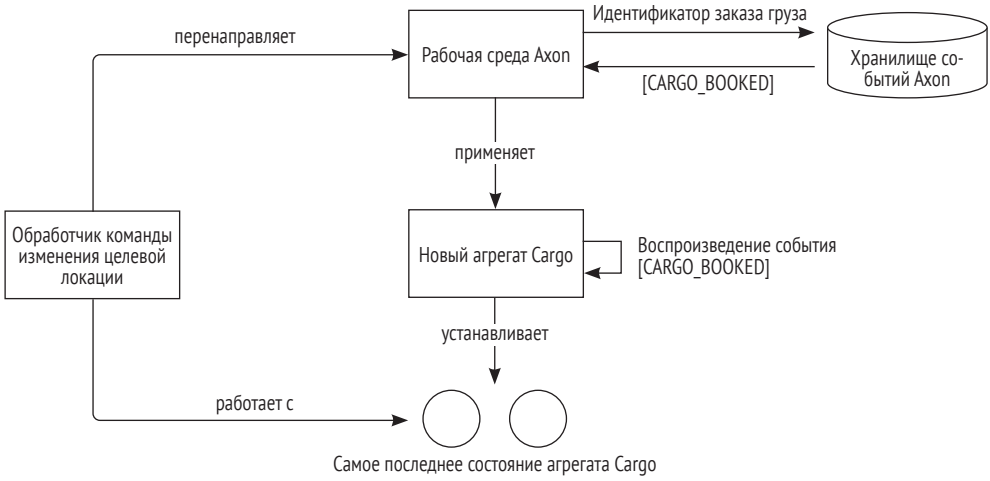

Рис. 6.42 ❖ Процесс извлечения и сопровождения состояния в рабочей среде Axon – команда изменения целевой локации до выполнения команды назначения маршрута

Полное графическое представление схемы потоков обработки событий показано на рис. 6.43.

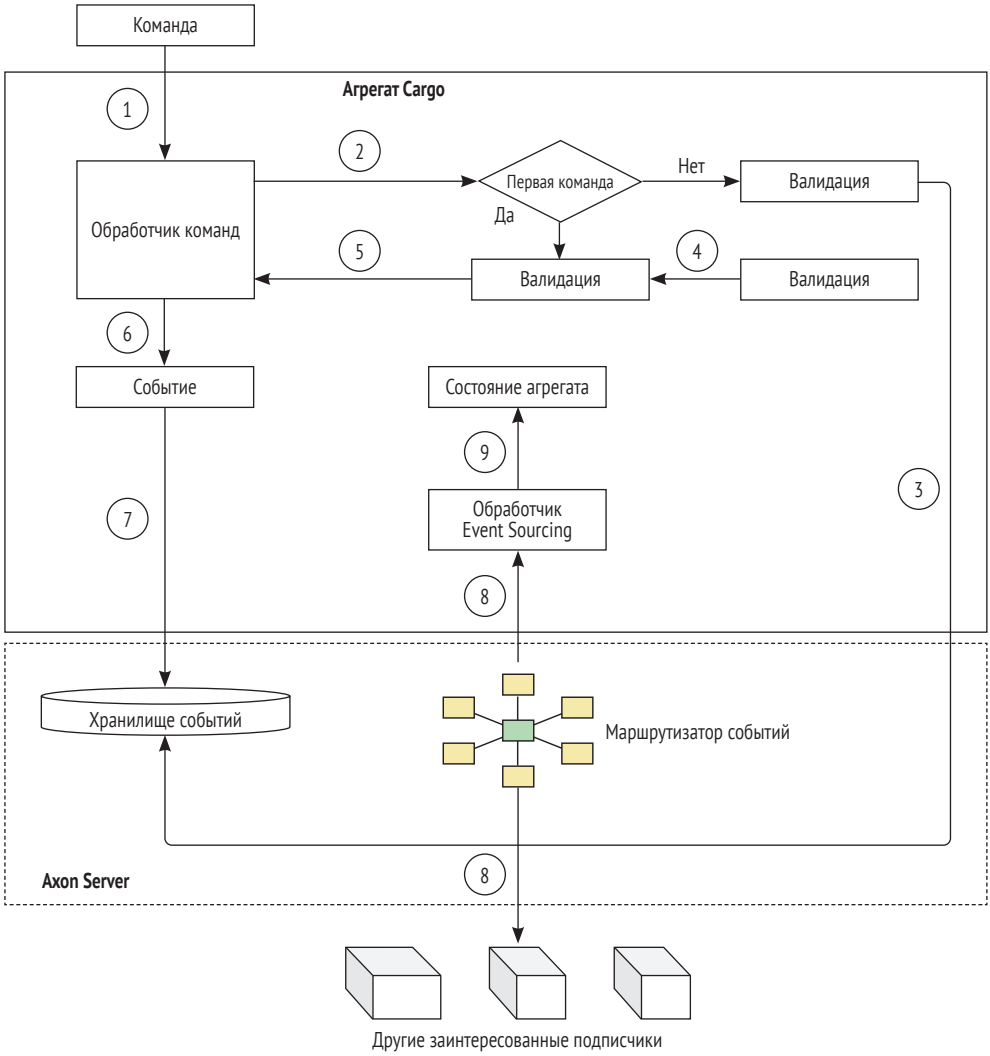

Рис. 6.43 ❖ Процесс извлечения и сопровождения состояния в рабочей среде Axon – для первой и всех последующих команд

На рис. 6.44 показана диаграмма класса для агрегата Cargo со всеми соответствующими событиями и обработчиками событий.

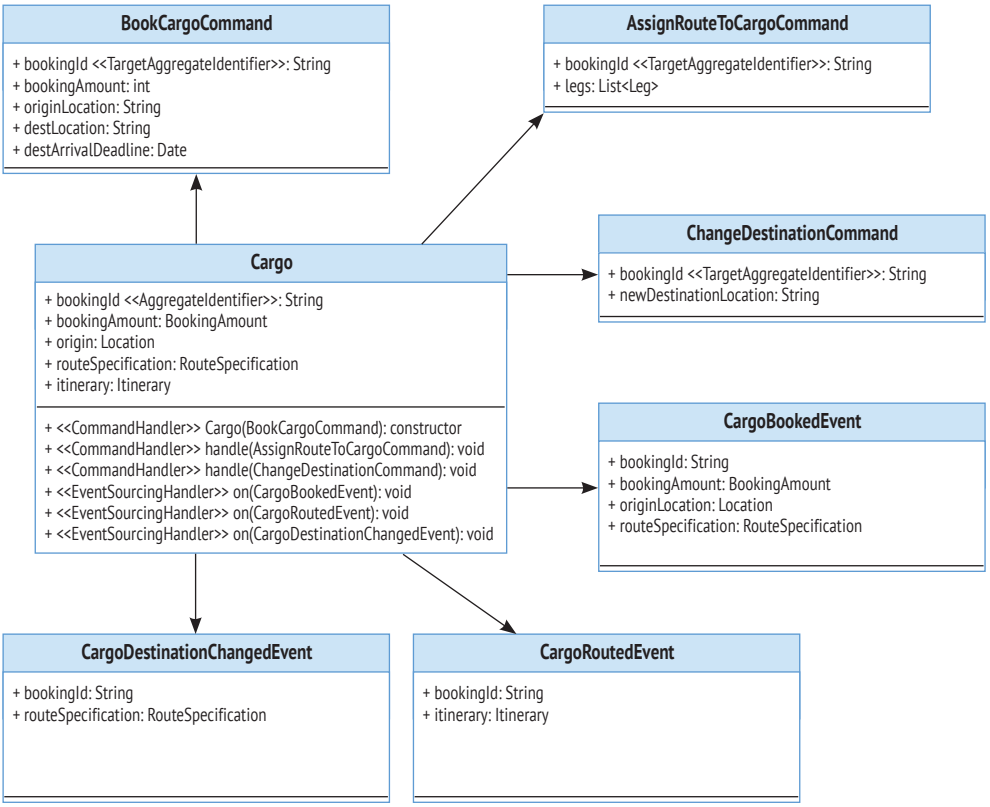

Рис. 6.44 ❖ Диаграмма класса для агрегата Cargo с соответствующими событиями и обработчиками событий

На этом завершается рассмотрение реализации агрегатов в ограниченном контексте Booking (заказ груза).

Проекции агрегатов

В предыдущем разделе рассматривалась полная реализация агрегата в области команд ограниченного контекста. В этой реализации можно видеть, что состояние агрегата не хранится непосредственно в базе данных. События, которые уже произошли в агрегате, сохраняются в специализированном хранилище событий (Event Store). Команды не являются единственными операциями в ограниченном контексте. В ограниченном контексте также должны выполняться запросы. Существуют требования, согласно которым необходимо запрашивать состояние агрегата, например веб-страница, которая предъявляет оператору обзорную информацию по конкретному грузу. Запрос к хранилищу событий и попытка воспроизведения событий для получения текущего состояния не является оптимальной операций, поэтому не рекомендуется к применению. Представьте себе агрегат, в котором произошло множество событий как часть его жизненного цикла. Воспроизведение каждого события в таком агрегате для получения его текущего состояния становится недопустимо дорогостоящей

процедурой. Необходим другой механизм, который позволял бы напрямую получать состояние агрегата в оптимальном режиме.

Для этой цели используются проекции агрегатов. Проще говоря, проекция агрегата (Aggregate Projection) – это представление состояния агрегата в различных формах, т. е. модель чтения (Read Model) состояния агрегата. Для любого агрегата возможно существование нескольких его проекций в зависимости от типа варианта использования проекции. Проекция агрегата всегда сохраняется в хранилище, которое содержит данные проекции. Это хранилище данных может быть обычной реляционной базой данных (например, MySQL), базой данных типа NoSQL (например, MongoDB) или даже хранилищем, размещаемым непосредственно в оперативной памяти (например, Elastic). Хранилище данных также зависит от типа варианта использования проекции.

Для хранилища данных проекции всегда поддерживается актуальность его данных при помощи подписки на события, генерируемых областью команд, и соответствующих обновлений собственных данных (см. ниже раздел об интерфейсах обработчика событий). Проекция предоставляет уровень запросов (Query layer), который могут использовать внешние потребители для получения данных проекции.

Общая схема потока обработки проекции показана на рис. 6.45.

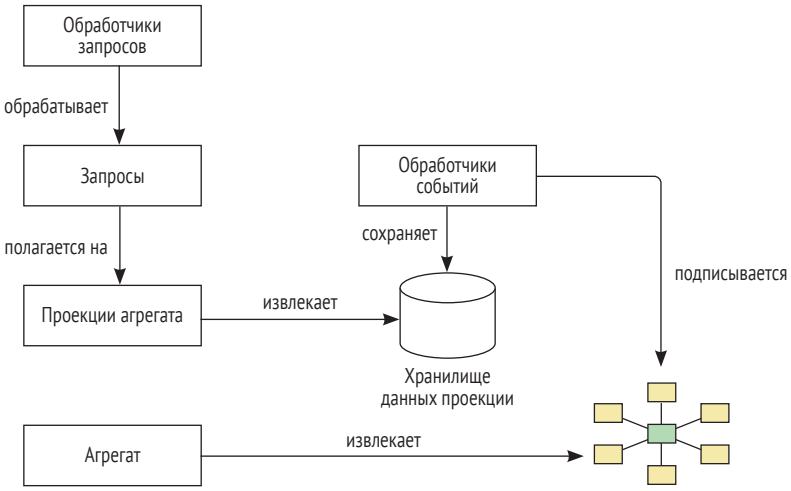

Рис. 6.45 ❖ Проекции в рабочей среде Axon

При реализации проекции агрегата охватываются следующие аспекты:

- реализация класса проекции агрегата;
- обработчик запроса;
- сопровождение состояния проекции.

Реализация класса проекции агрегата зависит от типа хранилища данных, которое выбрано для хранения состояния проекции. В приложении Cargo Tracker выбран вариант хранения состояния проекции в обычной реляционной базе данных MySQL.

Каждый из рассматриваемых здесь ограниченных контекстов будет иметь собственное хранилище проекций на основе MySQL. Каждая база данных может содержать несколько таблиц, в которых хранятся различные типы данных проекции в зависимости от типа варианта использования проекции. Классы проекций агрегата создаются на основе этих данных проекций, как показано на рис. 6.46.

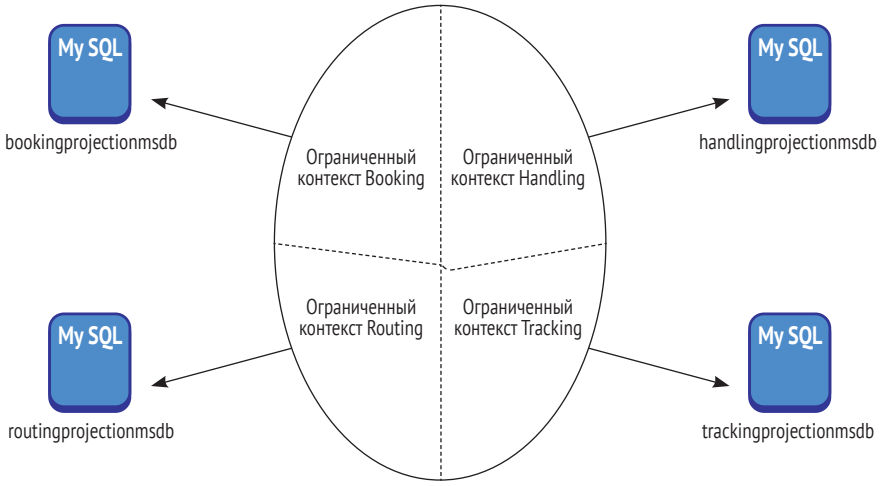

Рис. 6.46 ❖ Ограниченные контексты – базы данных проекций на основе СУБД MySQL

Так как выбранное хранилище данных является СУБД типа SQL, реализация класса проекции агрегата основана на JPA (Java Persistence API).

Рассмотрим подробнее реализацию класса проекции агрегата на примере проекции Cargo Summary (общая информация о грузе). Эта проекция использует таблицу *cargo_summary_projection* из базы данных MySQL с именем *bookingprojectionmsdb*.

Проекция должна предоставлять следующую подробную информацию о заказанном грузе:

- идентификатор заказа BookingId;
- состояние маршрута (назначен ли маршрут для этого груза или не назначен);
- состояние транспортировки (находится ли груз в порту или на судне);
- исходную локацию;
- целевую локацию (пункт назначения);
- предельный срок доставки.

Следует еще раз отметить, что требования к проекции могут отличаться в зависимости от варианта ее использования. Проекция Cargo Summary используется внешними потребителями для получения оперативной информации о том, что происходит с грузом в текущий момент.

В листинге 6.26 показана проекция Cargo Summary, реализованная как обычный объект JPA. Код реализации сохранен в модели предметной области (домена) в пакете projections.

Листинг 6.26 ❖ Агрегат Cargo как объект JPA

```
package com.practicalddd.cargotracker.bookingms.domain.projections;

import javax.persistence.*;
import java.util.Date;

/**
 * Projection class for the Cargo Aggregate implemented as a regular JPA
 * Entity. Contains a summary of the Cargo Aggregate
 */
@Entity   // Annotation to mark as a JPA Entity
@Table(name="cargo_summary_projection")   // Table Name Mapping
@NamedQueries({ //Named Queries
        @NamedQuery(name = "CargoSummary.findAll",
                query = "Select c from CargoSummary c"),
        @NamedQuery(name = "CargoSummary.findByBookingId",
                query = "Select c from CargoSummary c where c.booking_id =
:bookingId"),
        @NamedQuery(name = "Cargo.getAllBookingIds",
                query = "Select c.booking_id from CargoSummary c") })
public class CargoSummary {
    @Id
    private String booking_id;
    @Column
    private String transport_status;
    @Column
    private String routing_status;
    @Column
    private String spec_origin_id;
    @Column
    private String spec_destination_id;
    @Temporal(TemporalType.DATE)
    private Date deadline;
    protected CargoSummary(){
        this.setBooking_id(null);
    }
    public CargoSummary(String booking_id,
                        String transport_status,
                        String routing_status,
                        String spec_origin_id,
                        String spec_destination_id,
                        Date deadline){
        this.setBooking_id(booking_id);
        this.setTransport_status(transport_status);
        this.setRouting_status(routing_status);
        this.setSpec_origin_id(spec_origin_id);
        this.setSpec_destination_id(spec_destination_id);
        this.setDeadline(new Date());
    }
    public String getBooking_id() {   return booking_id;}
      public void setBooking_id(String booking_id) {this.booking_id =
booking_id;}
    public String getTransport_status() {return transport_status; }
```

```
      public void setTransport_status(String transport_status) { this.
transport_status = transport_status;}
    public String getRouting_status() {return routing_status;}
      public void setRouting_status(String routing_status) {this.routing_
status = routing_status; }
    public String getSpec_origin_id() { return spec_origin_id;  }
      public void setSpec_origin_id(String spec_origin_id) {this.spec_origin_
id = spec_origin_id; }
    public String getSpec_destination_id() {return spec_destination_id;}
      public void setSpec_destination_id(String spec_destination_id) {this.
spec_destination_id = spec_destination_id; }
    public Date getDeadline() { return deadline;}
    public void setDeadline(Date deadline) {this.deadline = deadline;}
}
```

На рис. 6.47 показана UML-диаграмма для класса проекции агрегата Cargo-Summary.

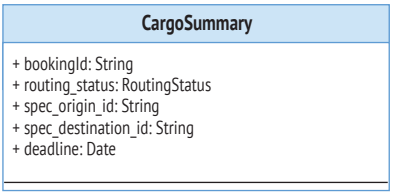

Рис. 6.47 ❖ Диаграмма класса проекции агрегата CargoSummary

После создания класса проекции, отображенного как объект JPA, и соответствующей таблицы базы данных можно переходить на уровень запросов.

Обработчики запросов

Запросы передаются в ограниченный контекст для извлечения состояния агрегата этого ограниченного контекста с помощью проекций агрегата. Реализация обработчика запросов включает следующие этапы:

- идентификацию и реализацию запросов;
- идентификацию и реализацию обработчиков запросов для обработки команд.

Идентификация запросов

Следует напомнить, что проекции агрегата представляют состояние этого агрегата. Проекции агрегата необходим уровень запросов, чтобы позволить внешним сторонам использовать (потреблять) данные проекции. Идентификация запросов включает любые операции, которым необходимы данные проекции агрегата.

Например, для ограниченного контекста Booking (заказ груза) существуют следующие требования, предъявляемые внешними потребителями через проекцию Cargo Summary, которая содержит необходимые данные для удовлетворения этих требований:

- общая обзорная информация о конкретном грузе;
- список информационных отчетов для всех грузов;
- список идентификаторов заказа груза для всех грузов.

Реализация запросов

Реализация идентифицированных запросов в рабочей среде Axon выполняется с использованием обычных простых старых объектов Java (POJO). Для каждого идентифицированного запроса необходимо реализовать класс `Query` и класс `QueryResult`. Класс `Query` – это сам запрос, который должен быть выполнен с учетом условий его выполнения. Класс `QueryResult` содержит результат выполнения соответствующего запроса.

Подробности реализации будут описаны на конкретных примерах. Рассмотрим запрос, который идентифицирован, как запрос на получение обзорной информации некоторого заказанного груза. Запрос получает имя `CargoSummaryQuery`, а класс результата выполнения этого запроса называется `CargoSummaryResult`.

В листинге 6.27 показан класс `CargoSummaryQuery`. Имя этого класса отображает его предназначение. В нем определен единственный конструктор, который принимает идентификатор заказа `BookingId`, т. е. критерий выполнения этого запроса.

Листинг 6.27 ❖ Реализация класса запроса CargoSummaryQuery

```
package com.practicalddd.cargotracker.bookingms.domain.queries;

/**
 * Реализация класса запроса Cargo Summary. Принимает идентификатор заказа груза BookingId,
 * который является критерием выполнения этого запроса.
 */
public class CargoSummaryQuery {
  private String bookingId;   // Критерий выполнения запроса.
  public CargoSummaryQuery( String bookingId ) {
    this.bookingId = bookingId;
  }
  @Override
  public String toString() { return "Cargo Summary for Booking Id" + bookingId; }
}
```

Здесь нет никаких затруднений – простой объект POJO, который содержит предназначение и критерий выполнения этого запроса.

Затем реализуется класс `CargoSummaryResult`, который содержит результат выполнения запроса, в данном случае это проекция `CargoSummaryProjection`, как показано в листинге 6.28.

Листинг 6.28 ❖ Реализация класса результата выполнения запроса CargoSummaryResult

```
package com.practicalddd.cargotracker.bookingms.domain.queries;

import com.practicalddd.cargotracker.bookingms.domain.projections.CargoSummary;

/**
 * Реализация класса результата запроса Cargo Summary, который содержит результат
```

```
 * выполнения запроса CargoSummaryQuery. Результат содержит данные из проекции
 * CargoSummary.
 */
public class CargoSummaryResult {
  private final CargoSummary cargoSummary;
  public CargoSummaryResult( CargoSummary cargoSummary ) { this.cargoSummary =
cargoSummary; }
  public CargoSummary getCargoSummary() { return cargoSummary; }
}
```

Теперь мы имеем реализацию класса запроса (`CargoSummary`), в котором определено предназначение запроса (получение обзора информации о грузе) и критерий запроса (идентификатор заказа груза).

Далее будет рассматриваться реализация обработки запроса.

Реализация обработчиков запросов

По аналогии с тем, как были реализованы обработчики команд, реализуются и обработчики запросов для обработки инструкций, содержащихся в запросах. Реализация обработчиков запросов включает идентификацию компонентов, которые могут обрабатывать запросы. В отличие от команд, размещаемых непосредственно в агрегатах, обработчики запросов располагаются в подпрограммах внутри обычных компонентов Spring Boot. Рабочая среда Axon предоставляет аннотацию с соответствующим наименованием `@QueryHandler` для обозначения подпрограмм в компонентах как обработчиков запросов.

В листинге 6.29 показана реализация обработчика `CargoAggregateQueryHandler`, который обрабатывает все запросы, имеющие отношение к проекции Cargo Summary для агрегата Cargo. В этом компоненте определяются два обработчика запросов: один – для обработки запроса `CargoSummaryQuery`, второй – для обработки запроса `ListCargoSummariesQuery`.

Ниже перечислены функции обработчиков запросов:

- прием запросов (`CargoSummaryQuery`, `ListCargoSummariesQuery`) как входных данных;
- выполнение именованных запросов JPA с использованием объекта JPA `CargoSummaryProjection`;
- возврат результатов обработки запроса (`CargoSummaryResult`, `ListCargoSummaryResult`).

В листинге 6.29 показана реализация обработчика `CargoAggregateQueryHandler`.

Листинг 6.29 ❖ Реализация обработчика CargoAggregateQueryHandler

```
package com.practicalddd.cargotracker.bookingms.domain.queryhandlers;

import com.practicalddd.cargotracker.bookingms.domain.projections.CargoSummary;
import com.practicalddd.cargotracker.bookingms.domain.queries.CargoSummaryQuery;
import com.practicalddd.cargotracker.bookingms.domain.queries.CargoSummaryResult;
import com.practicalddd.cargotracker.bookingms.domain.queries.ListCargoSummariesQuery;
import com.practicalddd.cargotracker.bookingms.domain.queries.ListCargoSummaryResult;

import org.axonframework.queryhandling.QueryHandler;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import org.springframework.stereotype.Component;

import javax.persistence.EntityManager;
import javax.persistence.Query;
import java.lang.invoke.MethodHandles;

/**
 * Класс, действующий как обработчик запросов для всех запросов,
 * относящихся к проекции Cargo Summary.
 */
@Component
public class CargoAggregateQueryHandler {
  private final static Logger logger =
                  LoggerFactory.getLogger( MethodHandles.lookup().lookupClass() );
  private final EntityManager entityManager;
  public CargoAggregateQueryHandler( EntityManager entityManager ) {
    this.entityManager = entityManager;
  }

  /**
   * Обработчик запроса, который возвращает проекцию Cargo Summary
   * по специализированному запросу.
   * @param cargoSummaryQuery
   * @return CargoSummaryResult
   */
  @QueryHandler
  public CargoSummaryResult handle( CargoSummaryQuery cargoSummaryQuery ) {
    logger.info("Handling {}", cargoSummaryQuery);
    Query jpaQuery =
       entityManager.createNamedQuery( "CargoSummary.findByBookingId", CargoSummary.class )
                  .setParameter( "bookingId", cargoSummaryQuery.getBookingId() );
    CargoSummaryResult result =
                     new CargoSummaryResult( (CargoSummary)jpaQuery.getSingleResult() );
    logger.info( "Returning {}", result );
    return result;
  }

  /**
   * Обработчик запроса, который возвращает все информационные отчеты Cargo.
   * @param listCargoSummariesQuery
   * @return CargoSummaryResult
   */
  @QueryHandler
  public ListCargoSummaryResult handle( ListCargoSummariesQuery listCargoSummariesQuery ) {
    logger.info( "Handling {}", listCargoSummariesQuery );
    Query jpaQuery = entityManager.createNamedQuery( "CardSummary.findAll",
                                                     CargoSummary.class );
    jpaQuery.setFirstResult( listCargoSummariesQuery.getOffset() );
    jpaQuery.setMaxResults( listCargoSummariesQuery.getLimit() );
    ListCargoSummaryResult result = new ListCargoSummaryResult( jpaQuery.getResultList() );
    return result;
  }
}
```

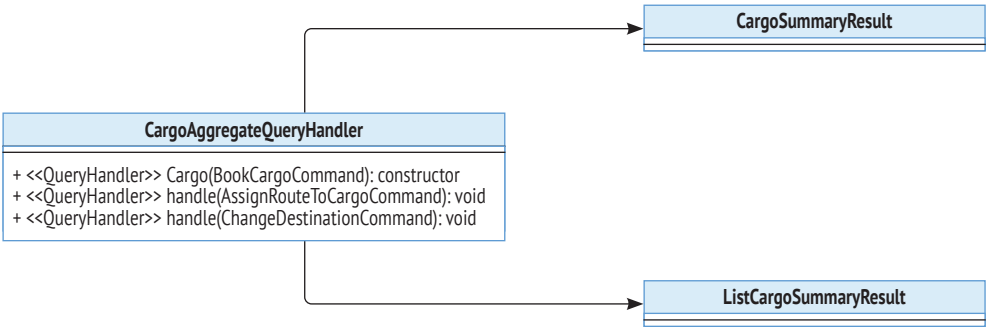

Рис. 6.48 ❖ Диаграмма класса обработчика запросов CargoAggregateQueryHandler

Необходимо сделать краткое замечание, прежде чем подвести итоги реализации проекций агрегатов.

Рабочая среда Axon предоставляет три типа реализации запросов:

- point-to-point – рассмотренные выше примеры были основаны на запросах типа point-to-point, где для каждого запроса существует соответствующий обработчик. Классы результатов в действительности завернуты в объект типа CompletableFuture<T> рабочей средой Axon, но это скрыто от разработчика;
- scatter-gather queries – этот тип запросов отправляется всем обработчикам, подписанным на данный запрос, а поток результатов возвращается и объединяется для передачи клиенту;
- subscription queries – это усовершенствованная функциональная возможность обработки запросов, предоставляемая рабочей средой Axon. Она позволяет клиенту получить начальное состояние проекции агрегата по запросу и поддерживать актуальность этого состояния с учетом изменения данных проекции, происходящих за определенный интервал времени.

На этом завершается рассмотрение реализации проекций агрегатов.

Саги

Завершающим этапом разработки модели предметной области (домена) является реализация саг. Как уже было отмечено ранее, саги можно реализовать двумя способами – на основе хореографии событий и на основе оркестровки событий.

Прежде чем перейти к подробностям реализации, необходимо снова обратиться к упрощенному представлению различных бизнес-потоков в приложении Cargo Tracker и рассмотреть саги в этих потоках.

На рис. 6.49 показаны бизнес-потоки и саги, которые являются частью этих потоков.

Сага Booking включает бизнес-операции в ограниченных контекстах Booking (заказ груза), Routing (назначение маршрута) и Tracking (отслеживание груза). Сага начинается с заказанного груза, потом назначается маршрут для этого груза, а завершается сага присваиванием идентификатора отслеживания

(Tracking Identifier) этому заказанному грузу. Идентификатор отслеживания используется клиентами для отслеживания процесса доставки груза.

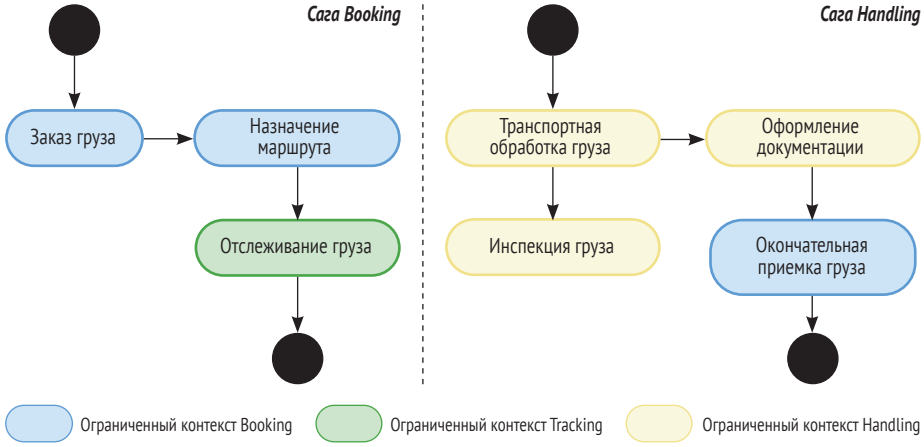

Рис. 6.49 ❖ Бизнес-потоки и саги, которые являются частью этих потоков

Сага Handling включает бизнес-операции Handling (транспортная обработка груза), Inspection (инспекция груза), Claims (оформление документации) и Settlement (окончательная приемка груза). Сага начинается с обрабатываемого груза в порту, в который он прибыл, затем клиент инспектирует груз и оформляет документацию. Завершается сага операцией окончательной приемки груза (например, выставляется иск за задержку доставки).

Обе эти саги можно реализовать на основе хореографии или оркестровки. Мы рассмотрим пример реализации саги Booking (заказ груза), но этот пример можно использовать и для реализации саги Handling (транспортная обработка груза). Мы сосредоточимся на варианте реализации с оркестровкой, используя встроенную поддержку рабочей среды Axon.

Прежде чем приступить непосредственно к реализации, необходимо подробно рассмотреть и уточнить различные команды, события и обработчики событий в саге Booking. Это показано на рис. 6.50.

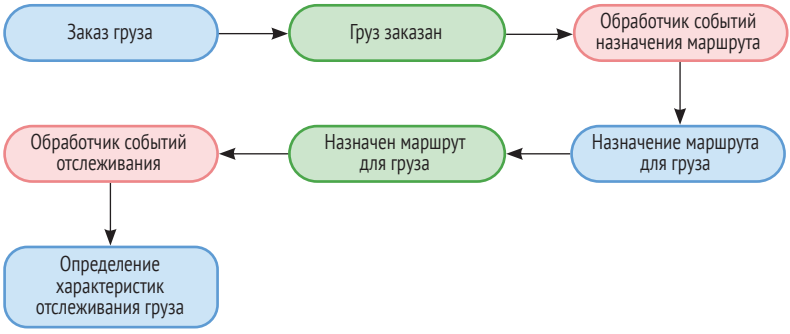

Рис. 6.50 ❖ Сага Booking (заказ груза)

По существу это представление реализации с использованием методики хореографии, при которой команды должны вызываться, события должны генерироваться, а обработчики должны обрабатывать события последовательно «по цепочке», пока не будет обработано самое последнее событие.

Реализация по методике оркестровки существенно отличается в том, что мы имеем центральный компонент, который обрабатывает события и последующие вызовы команд. Другими словами, ответственность за обработку событий и вызовы команд перемещается из отдельных обработчиков событий в центральный компонент, который выполняет эти действия.

На рис. 6.51 показана методика оркестровки.

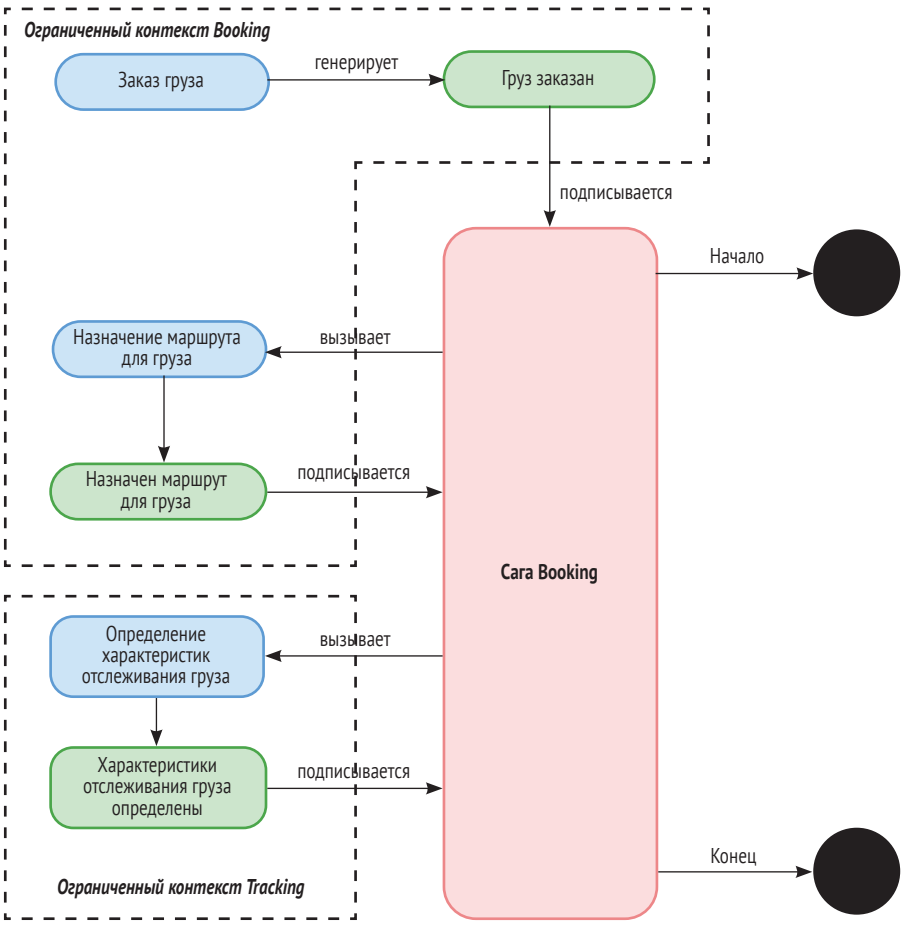

Рис. 6.51 ❖ Методика оркестровки

Рассмотрим подробнее отдельные этапы реализации:

- определяется имя саги. В рассматриваемом здесь примере для саги определяется имя Booking;
- после обработки команды заказа груза (Book Cargo) генерируется соответствующее событие `CargoBookedEvent`;

- сага Booking подписывается на событие `CargoBookedEvent` и начинает процесс саги;
- сага Booking отправляет на обработку инструкцию назначения маршрута `AssignRouteToCargoCommand`. Эта команда генерирует событие «маршрут назначен» `CargoRoutedEvent`;
- сага Booking подписывается на событие `CargoRoutedEvent`, затем отправляет на обработку инструкцию определения характеристик отслеживания груза `AssignTrackingDetailsToCargoCommand`. Эта команда генерирует событие «характеристики отслеживания груза определены» `TrackingDetailsAssignedEvent`;
- сага Booking подписывается на событие `TrackingDetailsAssignedEvent`, и, поскольку больше нет команд для обработки, принимает решение завершить сагу.

Здесь можно видеть, что центральный компонент – сага Booking – берет на себя всю ответственность за полную координацию и регулирование последовательностей команд и событий в нескольких ограниченных контекстах. Ни один из объектов моделей предметной области (домена) в этих ограниченных контекстах не знает о том, что он является участником процесса саги. Кроме того, эти объекты освобождены от обязательной подписки на события из других ограниченных контекстов с целью участия в транзакции. Они полагаются на централизованное выполнение всех этих действий сагой.

Сага на основе оркестровки представляет собой весьма мощный инструмент реализации распределенных транзакций в архитектуре микросервисов благодаря изначально присущей ей распределенной (несвязанной) природе, которая обеспечивает:

- полную изоляцию распределенных транзакций в специализированном компоненте;
- мониторинг и отслеживание потока распределенных транзакций;
- точную настройку и усовершенствование потока распределенных транзакций.

Исходный код саги реализуется с использованием разнообразных аннотаций, предоставляемых рабочей средой Axon. Реализация включает следующие шаги:

- используется обычный простой старый объект Java (POJO), который маркируется стандартной аннотацией `@Saga`, означающей, что этот класс действует как компонент сага;
- как уже отмечалось выше, сага отвечает за обработку событий и вызов команд. Рабочая среда Axon предоставляет специализированные для саг аннотации обработчиков событий `@SagaEventHandler`. Как и обычные обработчики событий, они размещаются в подпрограммах (методах) класса саги. Каждый обработчик событий саги должен быть обеспечен свойством ассоциации (association property). Это свойство позволяет рабочей среде Axon отобразить сагу в конкретном экземпляре агрегата, который участвует в этой саге;
- вызов команд выполняется стандартным для рабочей среды Axon способом, т. е. с использованием шлюза команд (Command Gateway);
- завершающий этап – реализация методов жизненного цикла в компоненте саги (начало работы, завершение работы). Рабочая среда Axon пре-

доставляет аннотацию `@StartSaga` для обозначения начала саги и метод `SagaLifecycle.end()` для завершения работы саги.

В листинге 6.30 показана реализация саги Booking.

Листинг 6.30 ❖ Реализация саги Booking

```
package   com.practicalddd.cargotracker.booking.application.internal.sagaparticipants;

import com.practicalddd.cargotracker.booking.application.internal.commandgateways
         .CargoBookingService;
import com.practicalddd.cargotracker.booking.domain.commands.AssignRouteToCargoCommand;
import com.practicalddd.cargotracker.booking.domain.commands
         .AssignTrackingDetailsToCargoCommand;
import com.practicalddd.cargotracker.booking.domain.events.CargoBookedEvent;
import com.practicalddd.cargotracker.booking.domain.events.CargoRoutedEvent;
import com.practicalddd.cargotracker.booking.domain.events.CargoTrackedEvent;

import org.axonframework.commandhandling.gateway.CommandGateway;
import org.axonframework.modelling.saga.SagaEventHandler;
import org.axonframework.modelling.saga.SagaLifecycle;
import org.axonframework.modelling.saga.StartSaga;
import org.axonframework.spring.stereotype.Saga;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.lang.invoke.MethodHandles;
import java.util.UUID;

/**
 * Менеджер саги Booking - реализация саги Booking.
 * Сага начинает работу, когда генерируется событие "груз заказан" (Cargo Booked).
 * Сага завершает работу после определения характеристик отслеживания груза.
 */
@Saga // Стандартная аннотация, определяющая, что это сага.
public class BookingSagaManager {
  private final static Logger logger =
                       LoggerFactory.getLogger( MethodHandles.lookup().lookupClass() );
  private CommandGateway commandGateway;
  private CargoBookingService cargoBookingService;
  /**
   * Зависимости для менеджера саги.
   * @param commandGateway
   */
  public BookingSagaManager( CommandGateway commandGateway,CargoBooking
                             Service cargoBookingService ) {
    this.commandGateway = commandGateway;
    this.cargoBookingService = cargoBookingService;
  }

  /**
```

```
   * Обработка события "груз заказан", начало работы саги и вызов команды назначения
   * маршрута для этого груза.
   * @param cargoBookedEvent
   */
  @StartSaga // Аннотация, обозначающая начало работы саги.
  @SagaEventHandler( associationProperty = "bookingId" )
  // Специальная аннотация саги для обработчика события.
  public void handle( CargoBookedEvent cargoBookedEvent ) {
    logger.info( "Handling the Cargo Booked Event within the Saga" );
    // Отправка команды назначения маршрута для груза.
    commandGateway.send( new AssignRouteToCargoCommand(cargoBookedEvent.getBookingId(),
                cargoBookingService.getLegs ForRoute( cargoBookedEvent.
getRouteSpecification() )
                        ) );
  }

  /**
   * Обработка события "маршрут назначен" и вызов команды определения характеристик
   * отслеживания груза.
   * @param cargoRoutedEvent
   */
  @SagaEventHandler( associationProperty = "bookingId" )
  public void handle( CargoRoutedEvent cargoRoutedEvent ) {
    logger.info( "Handling the Cargo Routed Event within the Saga" );
    String trackingId = UUID.randomUUID().toString();
    // Генерация случайного идентификатора отслеживания.
    SagaLifecycle.associateWith( "trackingId", trackingId );
    // Отправка команды определения характеристик отслеживания груза.
    commandGateway.send( new AssignTrackingDetailsToCargoCommand(
                            cargoRoutedEvent.getBookingId(),trackingId ) );
  }

  /**
   * Обработка события "характеристики отслеживания определены" и завершение работы саги.
   * @param cargoTrackedEvent
   */
  @SagaEventHandler( associationProperty = "trackingId" )
  public void handle( CargoTrackedEvent cargoTrackedEvent ) {
    SagaLifecycle.end();   // Завершение работы саги после обработки последнего события.
  }
}
```

Подведение итогов реализации

На этом завершается рассмотрение реализации модели предметной области (домена) с использованием рабочей среды Axon. На рис. 6.52 показана общая схема реализации.

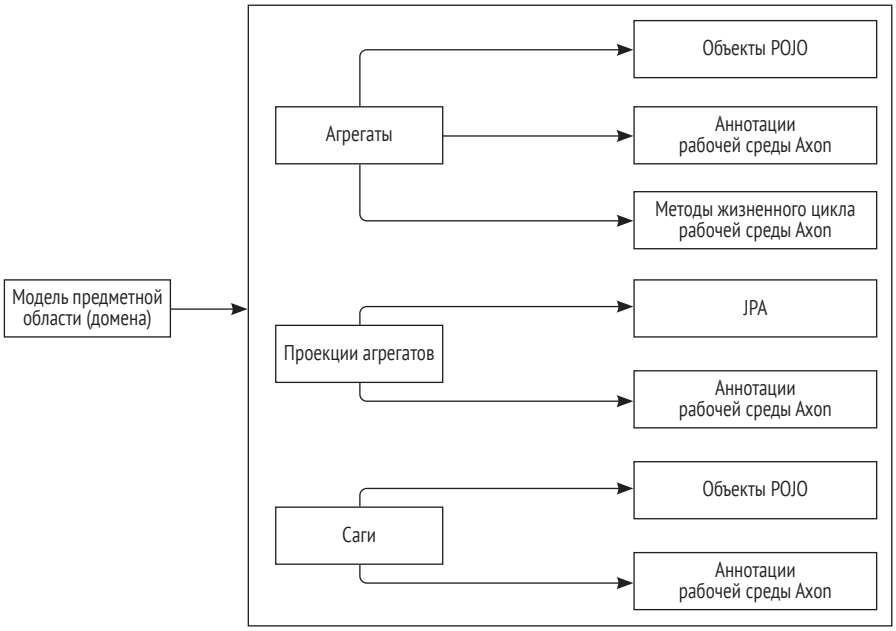

Рис. 6.52 ❖ Общая схема реализации модели предметной области (домена)

Реализация сервисов модели предметной области (домена) с использованием Axon

Следует напомнить, что сервисы модели предметной области (домена) обеспечивают поддержку сервисов модели предметной области (домена) (например, обеспечение возможности для внешних сторон использовать (потреблять) модель предметной области (домена), обеспечение обмена данными модели предметной области (домена) с внешними репозиториями). Реализация выполняется с использованием сочетания функциональных возможностей, предоставляемых платформой Spring Boot и рабочей средой Axon. Необходимо реализовать следующие типы сервисов модели предметной области (домена):

- входящие сервисы;
- сервисы приложения.

Входящие сервисы

Входящие сервисы (или входящие адаптеры в соответствии с терминологией гексагонального архитектурного шаблона) действуют как внешний шлюз для ядра модели предметной области (домена).

В приложении Cargo Tracker будут реализованы следующие входящие сервисы:

- уровень API на основе REST, который используется внешними потребителями для вызова операций в ограниченном контексте (команды и запросы);

- уровень обработки событий, реализованный рабочей средой Axon, который принимает (потребляет) события из шины событий и обрабатывает их.

REST API

REST API отвечает за прием от имени конкретного ограниченного контекста HTTP-запросов от внешних потребителей. Такой внешний запрос (request) может быть командой или запросом (query). Обязанность уровня REST API – преобразовать их в модель (формат) команд/запросов, распознаваемую моделью предметной области (домена) конкретного ограниченного контекста, и делегировать на уровень сервисов приложения для дальнейшей обработки.

На рис. 6.53 показана схема потоков и обязанностей REST API.

Рис. 6.53 ❖ Схема потоков и обязанностей REST API

Реализация REST API – это использование функциональных возможностей REST, предоставляемых платформой Spring Web. Как уже было отмечено выше в этой главе, эта зависимость была добавлена для приложений Spring Boot.

Рассмотрим пример реализации REST API. В листинге 6.31 показана реализация контроллера/REST API для команды заказа груза (Cargo Booking), выполняющая следующие функции:

- единственный внутренний метод POST принимает данные ресурса `BookCargoResource`, которые являются входной полезной нагрузкой для этого API;
- определение зависимости от `CargoBookingService`, который является сервисом приложения (будет подробно описан позже);
- преобразование данных ресурса `BookCargoResource` в модель (формат) команды `BookCargoCommand` с использованием класса утилиты Assembler (`BookCargoCommandDTOAssembler`);
- после преобразования выполняется делегирование процесса в сервис `CargoBookingService` для дальнейшей обработки.

Листинг 6.31 ❖ Реализация класса CargoBookingController

```
package com.practicalddd.cargotracker.bookingms.interfaces.rest;

import com.practicalddd.cargotracker.bookingms.interfaces.rest.transform
          .assembler.BookCargoCommandDTOAssembler;
import com.practicalddd.cargotracker.bookingms.interfaces.rest.transform.dto.
BookCargoResource;
import com.practicalddd.cargotracker.bookingms.application.internal.commandgateways
          .CargoBookingService;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;

/**
 * Интерфейс REST API для команды заказа груза.
 */
@RestController
@RequestMapping( "/cargobooking" )
public class CargoBookingController {
  private final CargoBookingService cargoBookingService;
  // Зависимость от сервиса приложения.
  /**
   * Обеспечение зависимостей.
   * @param cargoBookingService
   */
  public CargoBookingController( CargoBookingService cargoBookingService ) {
    this.cargoBookingService = cargoBookingService;
  }

  /**
   * Метод POST для заказа груза.
   * @param bookCargoCommandResource
   */
  @PostMapping
  @ResponseStatus( HttpStatus.CREATED )
  public void bookCargo( @RequestBody final BookCargoResource bookCargoCommandResource ) {
    cargoBookingService.bookCargo(
         BookCargoCommandDTOAssembler.toCommandFromDTO( bookCargoCommandResource ) );
  }
}
```

На этом завершается рассмотрение реализации входящих сервисов типа REST API.

Обработчик событий

Обработчики событий в ограниченном контексте отвечают за обработку событий, на которые подписан этот ограниченный контекст. Обработчики событий также отвечают для преобразования данных события в распознаваемую модель (формат) для дальнейшей обработки. Обработчики событий выполняют общее делегирование в сервисы приложения для обработки события после преобразования.

На рис. 6.54 показана схема потоков и обязанностей обработчика событий.

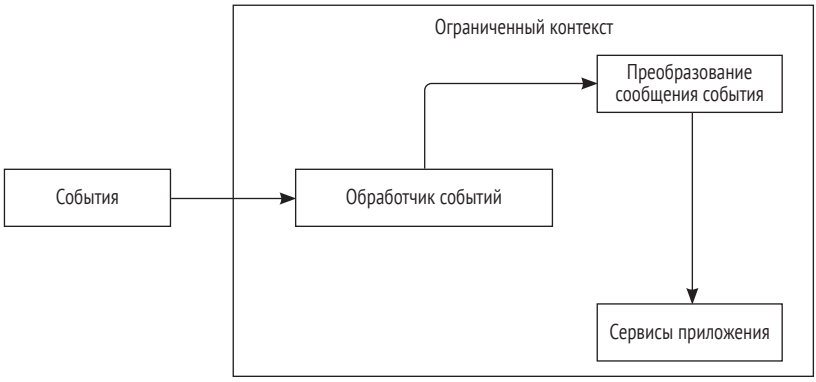

Рис. 6.54 ❖ Схема потоков и обязанностей обработчика событий

Реализация обработчиков событий выполняется с использованием стандартной аннотации `@EventHandler` рабочей среды Axon. Эти аннотации размещаются в подпрограммах обычных сервисов платформы Spring и содержат конкретные события, которые должен обрабатывать соответствующий обработчик событий.

Рассмотрим подробно пример реализации обработчика событий. В листинге 6.32 показан код обработчика события `CargoProjectionsEventHandler`. Этот обработчик события подписывается на события изменения состояния агрегата Cargo и соответствующим образом обновляет проекции агрегата Cargo (например, `CargoSummary`). Ниже описаны шаги реализации:

- класс обработчика событий маркируется аннотацией `@Service`;
- определяется зависимость от сервиса `CargoProjectionService`, который является одним из сервисов приложения (см. ниже);
- обеспечивается обработка события `CargoBookedEvent` посредством определения метода `handleCargoBookedEvent()` в классе обработчика с пометкой его аннотацией `@EventHandler`;
- метод `handleCargoBookedEvent()` использует событие `CargoBookedEvent` как полезную нагрузку события;
- выполняется преобразование данных события `CargoBookedEvent` в модель проекции агрегата `CargoSummary`;
- после преобразования обработчик делегирует процесс в сервис `CargoProjectionService` для дальнейшей обработки.

Листинг 6.32 ❖ Реализация обработчика события CargoProjectionsEventHandler

```
package com.practicalddd.cargotracker.bookingms.interfaces.events;

import com.practicalddd.cargotracker.bookingms.application.internal.CargoProjectionService;
import com.practicalddd.cargotracker.bookingms.domain.events.CargoBookedEvent;
import com.practicalddd.cargotracker.bookingms.domain.projections.CargoSummary;
import org.axonframework.eventhandling.EventHandler;
import org.axonframework.eventhandling.Timestamp;
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;

import javax.persistence.EntityManager;
import java.lang.invoke.MethodHandles;
import java.time.Instant;

/**
 * Обработчик всех событий, генерируемых агрегатом Cargo.
 */
@Service
public class CargoProjectionsEventHandler {
  private final static Logger logger =
                       LoggerFactory.getLogger( MethodHandles.lookup().lookupClass() );
  private CargoProjectionService cargoProjectionService;   // Зависимости.
  public CargoProjectionsEventHandler( CargoProjectionService cargoProjectionService ) {
    this.cargoProjectionService = cargoProjectionService;
  }

  /**
   * Обработчик события "груз заказан" (Cargo Booked). Выполняет преобразование данных
   * события в соответствующую модель проекции агрегата и делегирует эти данные в сервис
   * приложения для дальнейшей обработки.
   * @param cargoBookedEvent
   * @param eventTimestamp
   */
  @EventHandler
  public void cargoBookedEventHandler( CargoBookedEvent cargoBookedEvent,
                                       @Timestamp Instant eventTimestamp ) {
    logger.info( "Applying {}", cargoBookedEvent.getBookingId() );
    CargoSummary cargoSummary = new CargoSummary( cargoBookedEvent.getBookingId(),
                                                  "", "", "", "", new java.util.Date() );
    cargoProjectionService.storeCargoSummary( cargoSummary );
  }
}
```

На этом завершается рассмотрение реализации входящих сервисов.

Сервисы приложения

Сервисы приложения действуют как внешний связующий уровень или порт между входящими сервисами и моделью основной предметной области (домена) в ограниченном контексте. В приложении рабочей среды Axon в ограниченном контексте сервисы приложения отвечают за прием запросов (requests) от входящих сервисов и делегирование их соответствующим шлюзам, т. е. команды передаются в шлюз команд, а запросы передаются в шлюз запросов. После обработки событий результаты сохраняются (обеспечивается их персистентность) в зависимости от требуемых выходных данных (например, проекции сохраняются в специализированном хранилище данных).

На рис. 6.55 показаны обязанности сервисов приложения.

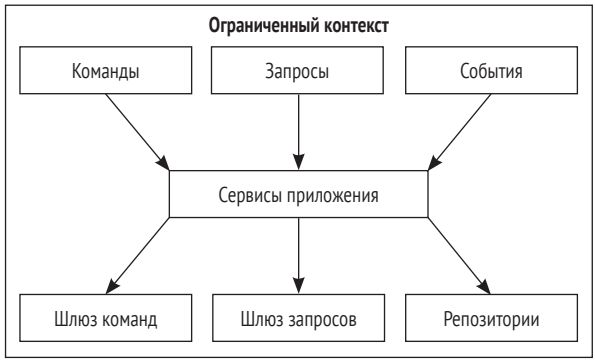

Рис. 6.55 ❖ Обязанности сервисов приложения

В листинге 6.33 показана реализация класса сервиса заказа груза, который отвечает за обработку всех команд, передаваемых в ограниченный контекст Booking.

Листинг 6.33 ❖ Реализация класса сервиса заказа груза

```
package com.practicalddd.cargotracker.bookingms.application.internal.commandgateways;

import com.practicalddd.cargotracker.bookingms.domain.commands.AssignRouteToCargoCommand;
import com.practicalddd.cargotracker.bookingms.domain.commands.BookCargoCommand;
import com.practicalddd.cargotracker.bookingms.domain.commands.ChangeDestinationCommand;
import org.axonframework.commandhandling.gateway.CommandGateway;
import org.springframework.stereotype.Service;

/**
 * Класс сервиса приложения для заказа груза, назначения маршрута для груза и для изменения
 * целевой локации (пункта назначения). Все команды, направленные в агрегат Cargo,
 * группируются в этом классе сервиса.
 */
@Service
public class CargoBookingService {
  private final CommandGateway commandGateway;
  public CargoBookingService( CommandGateway commandGateway ) {
    this.commandGateway = commandGateway;
  }

  /**
   * Заказ груза.
   * @param bookCargoCommand
   */
  public void bookCargo( BookCargoCommand bookCargoCommand ) {
    commandGateway.send( bookCargoCommand );   // Вызов шлюза команд (передача команды).
  }

  /**
```

```
   * Изменение целевой локации (пункта назначения) для груза.
   * @param changeDestinationCommand
   */
  public void changeDestinationOfCargo( ChangeDestinationCommand changeDestinationCommand
) {
    commandGateway.send( changeDestinationCommand );   // Вызов шлюза команд
                                                       // (передача команды).
  }

  /**
   * Назначение маршрута доставки груза.
   * @param assignRouteToCargoCommand
   */
  public void assignRouteToCargo( AssignRouteToCargoCommand assignRouteToCargoCommand ) {
    commandGateway.send( assignRouteToCargoCommand );   // Вызов шлюза команд (передача
                                                        // команды).
  }
}
```

На этом завершается рассмотрение реализации сервисов приложения и сервисов модели предметной области (домена).

Резюме

Краткое обобщение содержимого этой главы:

- в начале главы была приведена подробная информация о платформе Axon, в том числе о рабочей среде Axon и о сервере Axon Server;
- были рассмотрены все подробности процесса разработки и реализации различных артефактов предметно-ориентированного проектирования – в первую очередь модели предметной области (домена), включая агрегаты, команды и запросы с использованием платформы Spring Boot и рабочей среды Axon;
- были подробно рассмотрены все характеристики и свойства шаблона Event Sourcing, реализованного в рабочей среде Axon;
- завершилась глава подробным рассмотрением реализации сервисов модели предметной области (домена) с использованием функциональных возможностей, предоставляемых рабочей средой Axon.

Предметный указатель

E

J

С

У

Х

Ч

Ш

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Виджей Наир

Предметно-ориентированное проектирование в Enterprise Java с помощью Jakarta EE, Eclipse MicroProfile, Spring Boot и программной среды Axon Framework

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Снастин А. В.*
Корректор *Абросимова Л. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура PT Serif. Печать офсетная.
Усл. печ. л. 24,86. Тираж 200 экз.

Отпечатано в ООО «Принт-М»
142300, Московская обл., Чехов, ул. Полиграфистов, 1

Веб-сайт издательства: **www.dmkpress.com**