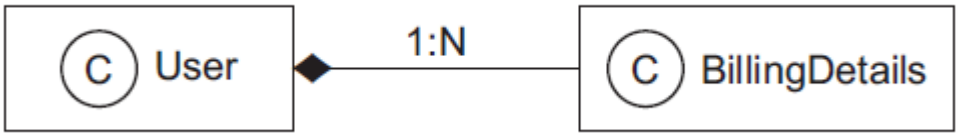
Разработка серверного ПО. Лекция 6. Начало работы с ORM

Фреймворк	Характеристики
JPA	+ Использует общий JPA API и требует наличия провайдера данных.
	+ Мы можем переключаться между провайдерами данных из конфигурации.
	+ Требуется явное управление EntityManagerFactory, EntityManager, и транзакциями.
	+ Конфигурация и объем кода, который необходимо написать, аналогичны нативному Hibernate.
	+ Мы можем перейти к JPA, создав EntityManagerFactory из нативного Hibernate.
Native Hibernate	+ Используется родной API Hibernate. Вы будете привязаны к использованию выбранного фреймворка.
	+ Создает свою конфигурацию в отдельных конфигурационных файлах Hibernate — по умолчанию (hibernate.cfg.xml или hibernate.properties).
	+ Требуется явное управление SessionFactory, сессиями и транзакциями.
	+ Конфигурация и объем кода, который необходимо написать, аналогичны подходу JPA.
	+ Мы можем перейти к нативному Hibernate, развернув Session-Factory из EntityManagerFactory или Session из EntityManager.
Spring Data JPA	+ Требует наличия в проекте дополнительных зависимостей Spring Data.
	+ Конфигурация позаботится о создании бинов, необходимых для проекта, включая менеджер транзакций.
	+ Интерфейс репозитория нужно только объявить, и Spring Data создаст его реализацию в виде прокси-класса со сгенерированными методами, которые взаимодействуют с базой данных.
	+ Необходимый репозиторий инжектируется и не создается программистом явно.
	+ Этот подход требует написания наименьшего количества кода, поскольку конфигурация берет на себя большую часть нагрузки.

Большинству приложений требуется персистентные хранилища данных. Когда мы говорим о персистентности в Java, мы обычно имеем в виду отображение и хранение экземпляров объектов в базе данных с помощью SQL.

Несоответствие объектной и реляционной парадигмы можно разбить на несколько частей, которые мы рассмотрим по очереди. Начнем исследование с простого примера, в котором нет никаких проблем. По мере того как мы будем его развивать, вы увидите, что несоответствие начинает проявляться.

Предположим, вам нужно разработать и реализовать приложение для электронной коммерции в Интернете. В этом приложении вам нужен класс для представления информации о пользователе системы, а другой класс - для представления информации о платежных реквизитах пользователя



На этой диаграмме видно, что у пользователя есть множество BillingDetails.Композиция - это тип связи, при которой объект (в нашем случае BillingDetails) не может концептуально существовать без контейнера (в нашем случае User). Вы можете перемещаться по отношениям между классами в обоих направлениях; это означает, что вы можете выполнять итерации в коллекциях или вызывать методы, чтобы добраться до «другой» стороны отношений. Классы, представляющие эти сущности, могут быть очень простыми:

```
public class User {
    private String username;
    private String address;
    private Set<BillingDetails> billingDetails = new HashSet<>();
    // Constructor, accessor methods (getters/setters), business methods
}

public class BillingDetails {
    private String account;
    private String bankname;
    private User user;
    // Constructor, accessor methods (getters/setters), business methods
}
```

Нас интересует только состояние сохранности сущностей, поэтому мы опустили реализацию конструкторов, методов-доступа и бизнес-методов. Для этого случая легко придумать схему SQL

```
CREATE TABLE USERS (
    USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,
    ADDRESS VARCHAR(255) NOT NULL
);

CREATE TABLE BILLINGDETAILS (
    ACCOUNT VARCHAR(15) NOT NULL PRIMARY KEY,
    BANKNAME VARCHAR(255) NOT NULL,
    USERNAME VARCHAR(15) NOT NULL,
    FOREIGN KEY (USERNAME) REFERENCES USERS(USERNAME)
);
```

Столбец USERNAME в BILLINGDETAILS, ограниченный внешним ключом, представляет отношения между двумя сущностями. Для этой простой доменной модели объектно-реляционное несоответствие

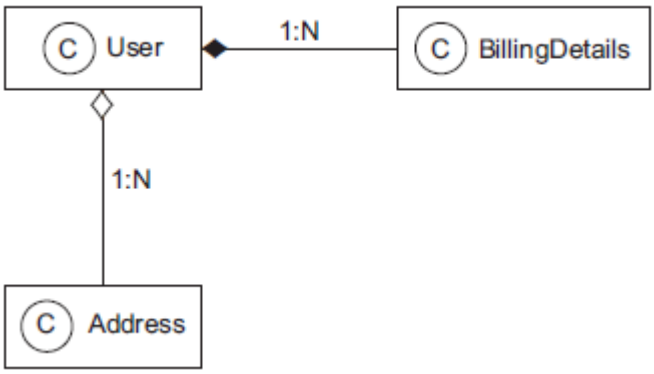
практически не заметно; можно легко написать JDBC-код для вставки, обновления и удаления информации о пользователях и биллинговых данных.

Теперь давайте посмотрим, что произойдет, если мы рассмотрим нечто более реалистичное.

Несоответствие отображения станет заметным, когда мы добавим в приложение больше сущностей и отношений между ними.

Проблема гранулярности

Самая очевидная проблема с текущей реализацией заключается в том, что мы создали адрес как простое строковое значение. В большинстве систем необходимо хранить информацию об улице, городе, штате, стране и почтовом индексе отдельно. Конечно, можно добавить эти свойства непосредственно в класс User, но поскольку другие классы в системе, скорее всего, также будут хранить информацию об адресе, разумнее создать класс Address, чтобы использовать ее повторно.



Отношение между User и Address называется агрегацией, на что указывает пустой ромб. Должны ли мы также добавить таблицу ADDRESS? Не обязательно; обычно адресная информация хранится в таблице USERS, в отдельных столбцах. Такая конструкция, скорее всего, будет работать лучше, потому что объединение таблиц не требуется, если вы хотите получить пользователя и адрес в одном запросе. Самым лучшим решением может быть создание нового типа данных SQL для представления адресов и добавление одного столбца этого нового типа в таблицу USERS, вместо добавления нескольких новых столбцов. Выбор между добавлением нескольких столбцов или одного столбца нового типа данных SQL - это проблема гранулярности. В широком смысле слова, гранулярность относится к относительному размеру типов, с которыми вы работаете. Давайте вернемся к примеру. Добавление нового типа данных в каталог базы данных для хранения экземпляров Address Java в одном столбце кажется наилучшим подходом:

```
CREATE TABLE USERS (
    USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,
    ADDRESS ADDRESS NOT NULL
);
```

Новый тип (класс) Address в Java и новый тип данных ADDRESS SQL должны гарантировать совместимость. Но вы обнаружите различные проблемы, если проверите поддержку пользовательских типов данных (UDT) в современных системах управления базами данных SQL.

Поддержка UDT - одно из нескольких так называемых объектно-реляционных расширений традиционного SQL. Сам по себе этот термин сбивает с толку, поскольку означает, что система управления базами данных имеет (или должна поддерживать) сложную систему типов данных. К сожалению, поддержка UDT является несколько непонятной особенностью большинства СУБД SQL, и она, конечно, не переносится между различными продуктами. Более того, стандарт SQL поддерживает пользовательские типы данных, но не очень хорошо.

Это ограничение не является виной реляционной модели данных. Можно считать, что неспособность стандартизировать столь важную часть функциональности - результат войны между поставщиками объектных и реляционных баз данных в середине 1990-х годов. Сегодня большинство инженеров признают, что продукты SQL имеют ограниченные системы типов - без вопросов. Даже при наличии сложной системы UDT в вашей СУБД SQL вы все равно будете дублировать объявления типов, записывая новый тип на Java и снова на SQL. Попытки найти лучшее решение для Java-пространства, например SQLJ, к сожалению, не увенчались успехом. Продукты СУБД редко поддерживают развертывание и выполнение Java-классов непосредственно в базе данных, а если поддержка и есть, то она, как правило, ограничена очень базовой функциональностью в повседневном использовании.

По этим и другим причинам использование UDT или Java-типов в базе данных SQL в настоящее время не является общепринятой практикой, и маловероятно, что вы встретите унаследованную схему, в которой широко используются UDT. Поэтому мы не можем и не будем хранить экземпляры нашего нового класса Address в одном новом столбце, имеющем тот же тип данных, что и Java-слой.

Прагматичное решение этой проблемы имеет несколько столбцов встроенных вендорных типов SQL (таких как булевы, числовые и строковые типы данных). Обычно таблицу USERS определяют следующим образом:

```
CREATE TABLE USERS (
    USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,
    ADDRESS_STREET VARCHAR(255) NOT NULL,
    ADDRESS_ZIPCODE VARCHAR(5) NOT NULL,
    ADDRESS_CITY VARCHAR(255) NOT NULL
);
```

Классы в доменной модели Java имеют различные уровни детализации: от грубых классов сущностей типа User до более тонких классов типа Address, вплоть до простых SwissZipCode, расширяющих AbstractNumericZipCode (или любой другой желаемый уровень абстракции). В отличие от этого, в базе данных SQL можно увидеть всего два уровня детализации типов: созданные вами типы отношений, такие как USERS и BILLINGDETAILS, и встроенные типы данных, такие как VARCHAR, BIGINT и TIMESTAMP.

Многие простые механизмы персистентности не распознают это несоответствие и в итоге вынуждают менее гибкое представление продуктов SQL накладываться на объектно-ориентированную модель, фактически уплощая ее. На самом деле, проблему гранулярности не так уж сложно решить, даже если она заметна во многих существующих системах.

Гораздо более сложная и интересная проблема возникает, когда мы рассматриваем модели домена, основанные на наследовании - свойстве объектно-ориентированного проектирования, которое вы

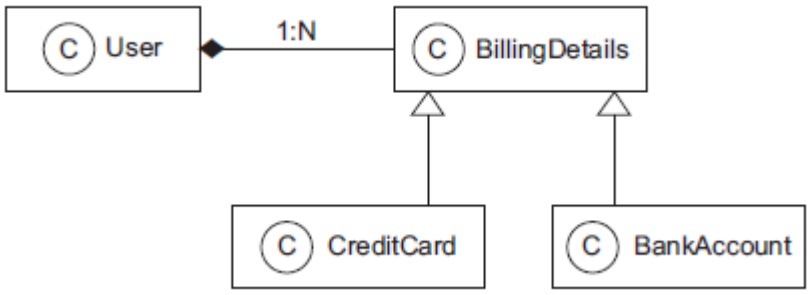
можете использовать для выставления счетов пользователям вашего приложения электронной коммерции новыми и интересными способами.

Проблема наследования

В Java наследование типов реализуется с помощью суперклассов и подклассов. Чтобы проиллюстрировать, почему это может создать проблему несоответствия, давайте изменим наше приложение электронной коммерции таким образом, чтобы теперь мы могли принимать не только оплату по банковским счетам, но и по кредитным картам.

Наиболее естественным способом отразить это изменение в модели является использование наследования для суперкласса BillingDetails, а также нескольких конкретных подклассов: CreditCard, BankAccount. Каждый из этих подклассов определяет немного разные данные (и совершенно разную функциональность, которая действует с этими данными). Диаграмма классов UML на рисунке 1.3 иллюстрирует эту модель.

Какие изменения мы должны внести, чтобы поддержать эту обновленную структуру классов Java? Можем ли мы создать таблицу CREDITCARD, которая расширяет BILLINGDETAILS? Продукты баз данных SQL обычно не реализуют наследование таблиц (или даже наследование типов данных), а если и реализуют, то не придерживаются стандартного синтаксиса.



Мы еще не закончили с наследованием. Как только мы вводим наследование в модель, у нас появляется возможность полиморфизма. Класс User имеет полиморфную ассоциацию с суперклассом BillingDetails. Во время выполнения экземпляр User может ссылаться на экземпляр любого из подклассов BillingDetails. Аналогично, мы хотим иметь возможность писать полиморфные запросы, которые ссылаются на класс BillingDetails, и чтобы запрос возвращал экземпляры его подклассов.

В базах данных SQL нет очевидного способа (или хотя бы стандартизированного способа) представления полиморфной ассоциации. Ограничение внешнего ключа ссылается ровно на одну целевую таблицу; определить внешний ключ, который ссылается на несколько таблиц, не так-то просто. В результате такого несоответствия подтипов структура наследования в модели должна храниться в базе данных SQL, которая не предлагает механизма наследования. К счастью, в настоящее время эта проблема хорошо изучена сообществом, и большинство решений поддерживают примерно одинаковую функциональность. Следующий аспект проблемы несоответствия объектов и реляций - это вопрос идентичности объектов.

Проблема идентичности объектов

В примере USERNAME определен как первичный ключ таблицы USERS. Был ли это хороший выбор? Как вы справляетесь с идентичными объектами в Java? Хотя проблема идентичности может быть

неочевидной поначалу, вы будете часто сталкиваться с ней, например, когда вам нужно будет проверить, идентичны ли два экземпляра. Существует три способа решения этой проблемы: два в мире Java и один в базе данных SQL. Как и ожидалось, они работают вместе только с некоторой помощью.

Java определяет два разных понятия идентичности:

- Идентичность экземпляра (примерно эквивалентна ячейке памяти, проверяется с помощью a == b)
- Равенство экземпляров, определяемое реализацией метода equals() (также называется равенством по значению).

С другой стороны, идентичность строки базы данных выражается как сравнение значений первичного ключа. Ни equals(), ни == не всегда эквивалентны сравнению значений первичных ключей. В Java часто бывает, что несколько неидентичных экземпляров одновременно представляют одну и ту же строку базы данных, например, в параллельно работающих потоках приложения. Кроме того, существуют некоторые тонкие сложности, связанные с корректной реализацией equals() для класса и пониманием того, когда это может понадобиться.

Давайте на примере обсудим еще одну проблему, связанную с идентификацией базы данных. В определении таблицы USERS первичным ключом является USERNAME. К сожалению, такое решение затрудняет изменение имени пользователя; необходимо обновить не только строку USERS, но и значения внешних ключей во (многих) строках BILLINGDETAILS. Чтобы решить эту проблему рекомендуется использовать суррогатные ключи во всех случаях, когда вы не можете найти хороший естественный ключ. Суррогатный ключ - это столбец первичного ключа, не имеющий значения для пользователя приложения - другими словами, ключ, который не представляется пользователю приложения. Его единственное назначение - идентифицировать данные внутри приложения.

Например, вы можете изменить определения таблиц следующим образом:

```
CREATE TABLE USERS (
    ID BIGINT NOT NULL PRIMARY KEY,
    USERNAME VARCHAR(15) NOT NULL UNIQUE,
    . . .
);
CREATE TABLE BILLINGDETAILS (
    ID BIGINT NOT NULL PRIMARY KEY,
    ACCOUNT VARCHAR(15) NOT NULL,
    BANKNAME VARCHAR(255) NOT NULL,
    USER_ID BIGINT NOT NULL,
    FOREIGN KEY (USER_ID) REFERENCES USERS(ID)
);
```

Столбцы ID содержат значения, генерируемые системой. Эти столбцы были введены исключительно для пользы модели данных.

В контексте персистентности идентификация тесно связана с тем, как система обрабатывает кэширование и транзакции. Различные решения для персистентности выбирают разные стратегии, и это является довольно сложной темой.

Пока что скелетное приложение электронной коммерции, которое мы разработали, выявило проблемы несоответствия парадигм из-за гранулярности отображения, подтипов и проблем идентичности. Далее нам необходимо обсудить важную концепцию ассоциаций: как отображаются и обрабатываются отношения между сущностями. Является ли ограничение внешнего ключа в базе данных всем, что вам нужно?

Проблема ассоциаций

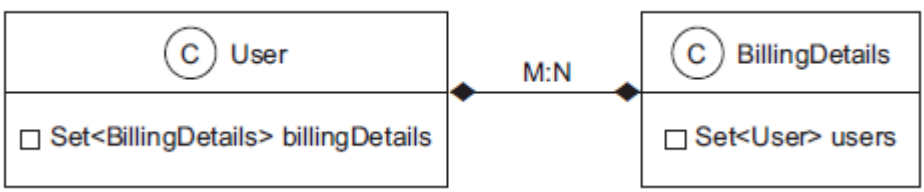
В доменной модели ассоциации представляют собой отношения между сущностями. Классы User, Address и BillingDetails ассоциированы, но, в отличие от Address, BillingDetails существует сам по себе. Экземпляры BillingDetails хранятся в собственной таблице. Сопоставление и управление ассоциациями сущностей - центральные понятия в любом решении по сохранению объектов.

Объектно-ориентированные языки представляют ассоциации с помощью ссылок на объекты, но в реляционном мире столбец с ограничением по внешнему ключу представляет собой ассоциацию с копиями значений ключей. Ограничение - это правило, гарантирующее целостность ассоциации. Между этими двумя механизмами есть существенные различия. Ссылки на объекты по своей сути являются направленными; ассоциация идет от одного экземпляра к другому. Они являются указателями. Если же ассоциация между экземплярами должна быть проходимой в обоих направлениях, вы должны определить ассоциацию дважды, по одному разу в каждом из связанных классов.


```
public class User {
    private Set<BillingDetails> billingDetails = new HashSet<>();
}

public class BillingDetails {
    private User user;
}
```

Навигация в определенном направлении не имеет смысла для реляционной модели данных, поскольку вы можете создавать ассоциации данных с помощью операторов join и projection. Задача состоит в том, чтобы сопоставить полностью открытую модель данных, не зависящую от приложения, которое работает с данными, с навигационной моделью, зависящей от приложения, - ограниченным представлением ассоциаций, необходимых данному конкретному приложению.



```
public class User {
    private Set<BillingDetails> billingDetails = new HashSet<>();
}

public class BillingDetails {
    private Set<User> users = new HashSet<>();
}
```

Однако объявление внешнего ключа в таблице BILLINGDETAILS представляет собой ассоциацию «многие-ко-многим»: каждый банковский счет связан с определенным пользователем, но у каждого пользователя может быть несколько связанных банковских счетов. Если вы хотите представить в базе данных SQL ассоциацию «многие-ко-многим», вы должны ввести новую таблицу, обычно называемую таблицей связей. В большинстве случаев эта таблица не появляется нигде в доменной модели. В данном примере, если вы считаете, что отношения между пользователем и информацией о счетах являются отношениями «многие-ко-многим», вы должны определить таблицу связей следующим образом:

```
CREATE TABLE USER_BILLINGDETAILS (
    USER_ID BIGINT,
    BILLINGDETAILS_ID BIGINT,
    PRIMARY KEY (USER_ID, BILLINGDETAILS_ID),
    FOREIGN KEY (USER_ID) REFERENCES USERS(ID),
    FOREIGN KEY (BILLINGDETAILS_ID) REFERENCES BILLINGDETAILS(ID)
);
```

До сих пор проблемы, которые мы рассматривали, были в основном структурными: вы можете увидеть их, рассматривая чисто статический вид системы. Возможно, самая сложная проблема в сохранении объектов - это динамическая проблема: как осуществляется доступ к данным во время выполнения.

Проблема навигации по данным

Существует фундаментальное различие между доступом к данным в коде Java и в реляционной базе данных. В Java, когда вы получаете доступ к информации о счетах пользователя, вы вызываете someUser.getBillingDetails().iterator().next() или что-то подобное. Или, начиная с Java 8, вы можете вызвать someUser.getBillingDetails().stream()
.filter(someCondition).map(someMapping).forEach(billingDetails-> {doSomething (billingDetails)}). Это самый естественный способ доступа к объектно-ориентированным данным, который часто называют «хождением по объектной сети». Вы переходите от одного экземпляра к другому, даже итерируете коллекции, следуя подготовленным указателям между классами. К сожалению, это не самый эффективный способ получения данных из базы данных SQL. Самое важное, что вы можете сделать для повышения производительности кода доступа к данным, - это минимизировать количество запросов к базе данных. Самый очевидный способ сделать это - минимизировать количество SQL-запросов. (Конечно, другие, более сложные способы, такие как обширное кэширование, следуют в качестве второго шага). Поэтому для эффективного доступа к реляционным данным с помощью SQL обычно требуются соединения между интересующими таблицами. Количество таблиц, включаемых в объединение при получении данных, определяет глубину объектной сети, по которой можно

перемещаться в памяти. Например, если вам нужно получить данные о пользователе и вас не интересует информация о его счетах, вы можете написать такой простой запрос:

```
SELECT * FROM USERS WHERE ID = 123
```

С другой стороны, если вам нужно получить пользователя, а затем посетить каждый из связанных с ним экземпляров BillingDetails (скажем, чтобы перечислить банковские счета пользователя), вы напишете другой запрос:

```
SELECT * FROM USERS, BILLINGDETAILS
WHERE USERS.ID = 123 AND
BILLINGDETAILS.ID = USERS.ID
```

Как видите, для эффективного использования соединений необходимо знать, к какой части объектной сети вы планируете получить доступ, еще до того, как вы начнете перемещаться по объектной сети! Однако будьте осторожны: если вы получаете слишком много данных (возможно, больше, чем вам нужно), вы тратите память на уровне приложений. Кроме того, вы можете перегрузить базу данных SQL огромными наборами результатов декартова произведения. Представьте, что в одном запросе можно получить не только пользователей и банковские счета, но и все заказы, оплаченные с каждого банковского счета, товары в каждом заказе и так далее. Любое решение для сохранения объектов позволяет получать данные об ассоциированных экземплярах только при первом обращении к ассоциации в Java-коде. Это известно как ленивая загрузка: получение данных только по требованию. Такой фрагментарный стиль доступа к данным принципиально неэффективен в контексте базы данных SQL, поскольку требует выполнения одного оператора для каждого узла или коллекции объектной сети, к которой осуществляется доступ. Это страшная проблема n+1 selects. В нашем примере вам потребуется один select для получения пользователя, а затем n select для каждого из n связанных с ним экземпляров BillingDetails. Это несоответствие в способах доступа к данным в Java-коде и в реляционной базе данных является, пожалуй, единственным наиболее распространенным источником проблем с производительностью в информационных системах на Java. Избежать проблем с декартовым произведением и n+1 selects все еще остается проблемой для многих Java-программистов. Hibernate предоставляет сложные возможности для эффективного и прозрачного извлечения сетей объектов из базы данных в приложение, обращающееся к ним.

ORM, JPA, Hibernate и Spring Data

В двух словах, объектно-реляционное отображение (ORM) - это автоматизированное (и прозрачное) сохранение объектов Java-приложения в таблицах RDBMS (системы управления реляционными базами данных) с помощью метаданных, описывающих соответствие между классами приложения и схемой базы данных SQL. По сути, ORM работает путем преобразования (обратимого) данных из одного представления в другое. Программа, использующая ORM, предоставляет метаинформацию о том, как отобразить объекты из памяти в базу данных, а эффективное преобразование выполняет ORM. Некоторые люди могут считать одним из преимуществ ORM то, что он ограждает разработчиков от грязного SQL. Согласно этой точке зрения, от объектно-ориентированных разработчиков не следует ожидать глубокого погружения в SQL или реляционные базы данных. Напротив, Java-разработчики

должны быть достаточно хорошо знакомы с реляционным моделированием и SQL, чтобы работать с Hibernate и Spring Data. ORM - это продвинутая техника, используемая разработчиками, которые уже прошли этот сложный путь. JPA (Jakarta Persistence API, ранее Java Persistence API) - это спецификация, определяющая API, который управляет сохранением объектов и объектно-реляционных отображений. Hibernate - самая популярная реализация этой спецификации. Таким образом, JPA определяет, что нужно сделать для сохранения объектов, а Hibernate - как это сделать. Spring Data Commons, как часть семейства Spring Data, предоставляет основные концепции фреймворка Spring, которые поддерживают все модули Spring Data. Spring Data JPA, еще один проект из семейства Spring Data, представляет собой дополнительный слой поверх реализаций JPA (таких как Hibernate). Spring Data JPA не только может использовать все возможности JPA, но и добавляет свои собственные возможности, такие как генерация запросов к базе данных на основе имен методов. Чтобы эффективно использовать Hibernate, вы должны уметь просматривать и интерпретировать выдаваемые им SQL-запросы и понимать их последствия для производительности. Чтобы воспользоваться преимуществами Spring Data, вы должны уметь предвидеть, как создается код шаблонов и генерируемые запросы. Спецификация JPA определяет следующее

- Средство для указания метаданных отображения - того, как постоянные классы и их свойства соотносятся со схемой базы данных. JPA в значительной степени опирается на Java-аннотации в классах доменной модели, но вы также можете писать отображения в XML-файлах.
- API для выполнения базовых CRUD-операций над экземплярами постоянных классов, в первую очередь avax.persistence.EntityManager для хранения и загрузки данных.
- Язык и API для определения запросов, которые ссылаются на классы и свойства классов. Этот язык - Jakarta Persistence Query Language (JPQL), и он похож на SQL. Стандартизированный API позволяет программно создавать критериальные запросы без манипуляций со строками.
- Как механизм персистентности взаимодействует с транзакционными экземплярами для выполнения проверки на загрязненность, поиска ассоциаций и других функций оптимизации. В спецификации JPA описаны некоторые базовые стратегии кэширования.

Hibernate реализует JPA и поддерживает все стандартизированные отображения, запросы и программные интерфейсы. Давайте рассмотрим некоторые преимущества Hibernate:

- Производительность - Hibernate устраняет большую часть повторяющейся работы (больше, чем вы ожидаете) и позволяет вам сосредоточиться на бизнес-проблеме. Независимо от того, какую стратегию разработки приложений вы предпочитаете - «сверху вниз» (начиная с модели домена) или «снизу вверх» (начиная с существующей схемы базы данных), - Hibernate, используемый вместе с соответствующими инструментами, значительно сократит время разработки.

Удобство обслуживания - автоматизированная ORM с помощью Hibernate сокращает количество строк кода, делая систему более понятной и легкой для рефакторинга. Hibernate обеспечивает буфер между моделью домена и схемой SQL, изолируя каждую модель от незначительных изменений в другой.

- Производительность - хотя персистентность, созданная вручную, может быть быстрее в том же смысле, в каком код на ассемблере может быть быстрее кода на Java, автоматизированные решения, такие как Hibernate, позволяют использовать множество оптимизаций в любое время. Одним из примеров является эффективное и легко настраиваемое кэширование на уровне приложений. Это означает, что разработчики могут потратить больше энергии на ручную

оптимизацию немногих оставшихся реальных узких мест, вместо того чтобы преждевременно оптимизировать все подряд.

- Независимость от поставщиков - Hibernate может помочь снизить некоторые риски, связанные с привязкой к поставщикам. Даже если вы планируете никогда не менять продукт СУБД, инструменты ORM, поддерживающие несколько различных СУБД, обеспечивают определенный уровень переносимости. Кроме того, независимость от СУБД помогает в сценариях разработки, когда инженеры используют легкую локальную базу данных, но развертывают ее для тестирования и производства на другой системе.

Spring Data делает реализацию слоя персистентности еще более эффективной. Spring Data JPA, один из проектов семейства, располагается поверх слоя JPA. Spring Data JDBC, еще один проект семейства, располагается поверх JDBC. Давайте рассмотрим некоторые преимущества Spring Data:

- Общая инфраструктура - Spring Data Commons, часть зонтичного проекта Spring Data, предоставляет модель метаданных для хранения Java-классов и технологически нейтральные интерфейсы репозиториев. Он предоставляет свои возможности другим проектам Spring Data.

- Удаление реализаций DAO Реализации JPA используют паттерн объекта доступа к данным (DAO). Этот паттерн основан на идее абстрактного интерфейса к базе данных и отображает вызовы приложения на слой персистентности, скрывая детали базы данных. Spring Data JPA позволяет полностью отказаться от реализации DAO, поэтому код будет короче.

- Автоматическое создание классов - при использовании Spring Data JPA интерфейс DAO должен расширять специфический для JPA интерфейс Repository - JpaRepository. Spring Data JPA автоматически создаст реализацию для этого интерфейса - программисту не придется заботиться об этом.

- Реализации по умолчанию для методов-Spring Data JPA будет генерировать реализации по умолчанию для каждого метода, определенного интерфейсами репозитория. Базовые операции CRUD больше не нужно реализовывать. Это сокращает объем шаблонного кода, ускоряет разработку и устраняет возможность внесения ошибок.

- Генерируемые запросы - вы можете определить метод в интерфейсе вашего репозитория, следуя шаблону именования. Нет необходимости писать запросы вручную; Spring Data JPA сам разберет имя метода и создаст для него запрос.

- Близость к базе данных при необходимости - Spring Data JDBC может взаимодействовать с базой данных напрямую, избегая «магии» Spring Data JPA. Он позволяет взаимодействовать с базой данных через JDBC, но избавляет вас от шаблонного кода, используя средства фреймворка Spring.

Введение в Hibernate

Объектно-реляционное отображение (ORM) - это технология программирования, позволяющая установить связь между несовместимыми мирами объектно-ориентированных систем и реляционных баз данных. Hibernate - это амбициозный проект, целью которого является полное решение проблемы управления постоянными данными в Java. Сегодня Hibernate - это не только сервис ORM, но и набор инструментов управления данными, выходящих далеко за рамки ORM.

Набор проектов Hibernate включает в себя следующее:

- Hibernate ORM - Hibernate ORM состоит из ядра, базового сервиса для доступа к базам данных SQL и собственного проприетарного API. Hibernate ORM является основой для нескольких других проектов пакета, и это самый старый проект Hibernate. Вы можете использовать Hibernate ORM самостоятельно, независимо от фреймворка или конкретной среды выполнения со всеми JDK. Если источник данных доступен, вы можете настроить его для Hibernate, и он будет работать.
- Hibernate EntityManager - это реализация Hibernate стандартного Jakarta Persistence API. Это дополнительный модуль, который можно установить поверх Hibernate ORM. Родные функции Hibernate во всех отношениях превосходят функции персистентности JPA.
- Hibernate Validator-Hibernate предоставляет эталонную реализацию спецификации Bean Validation (JSR 303). Независимо от других проектов Hibernate, он обеспечивает декларативную проверку для классов доменной модели (или любых других).
- Hibernate Envers-Envers предназначен для ведения журнала аудита и хранения нескольких версий данных в базе данных SQL. Это помогает добавить историю данных и контрольные записи в приложение, аналогично системам контроля версий.
- Hibernate Search-Hibernate Search поддерживает индекс данных доменной модели в актуальном состоянии в базе данных Apache Lucene. Он позволяет запрашивать эту базу данных с помощью мощного и естественно интегрированного API. Многие проекты используют Hibernate Search в дополнение к Hibernate ORM, добавляя возможности полнотекстового поиска. Если в пользовательском интерфейсе вашего приложения есть форма свободного текстового поиска, работайте с Hibernate Search.
- Hibernate OGM - этот проект Hibernate является объектным/грид-маппером. Он обеспечивает поддержку JPA для решений NoSQL, используя основной движок Hibernate, но сохраняя отображенные сущности в хранилищах данных, ориентированных на ключи/значения, документы или графы.
- Hibernate Reactive-Hibernate Reactive - это реактивный API для Hibernate ORM, взаимодействующий с базой данных в неблокирующей манере. Он поддерживает неблокирующие драйверы баз данных.

Введение в Spring Data

Spring Data - это семейство проектов, входящих в состав фреймворка Spring, целью которых является упрощение доступа к реляционным и NoSQL базам данных:

- Spring Data Commons-Spring Data Commons, часть зонтичного проекта Spring Data, предоставляет модель метаданных для хранения Java-классов и интерфейсы репозиториев, нейтральные к технологиям.
- Spring Data JPA-Spring Data JPA занимается реализацией репозиториев на основе JPA. Он обеспечивает улучшенную поддержку слоев доступа к данным на основе JPA за счет сокращения шаблонного кода и создания реализаций для интерфейсов репозиториев.
- Spring Data JDBC-Spring Data JDBC занимается реализацией репозиториев на основе JDBC. Он обеспечивает улучшенную поддержку уровней доступа к данным на основе JDBC. Он не

предлагает ряд возможностей JPA, таких как кэширование или ленивая загрузка, в результате чего получается более простой и ограниченный ORM.

- Spring Data REST-Spring Data REST занимается экспортом репозиториев Spring Data в виде REST-ресурсов.
- Spring Data MongoDB-Spring Data MongoDB занимается доступом к базе данных MongoDB. Он опирается на слой доступа к данным через репозиторий и модель программирования POJO.
- Spring Data Redis-Spring Data Redis занимается доступом к базе данных ключей/значений Redis. Он основан на освобождении разработчика от управления инфраструктурой и предоставлении высоко- и низкоуровневых абстракций для доступа к хранилищу данных.

"Hello World" с JPA

Пример в JPAHelloWorld. Класс JPAHelloWorldTest.

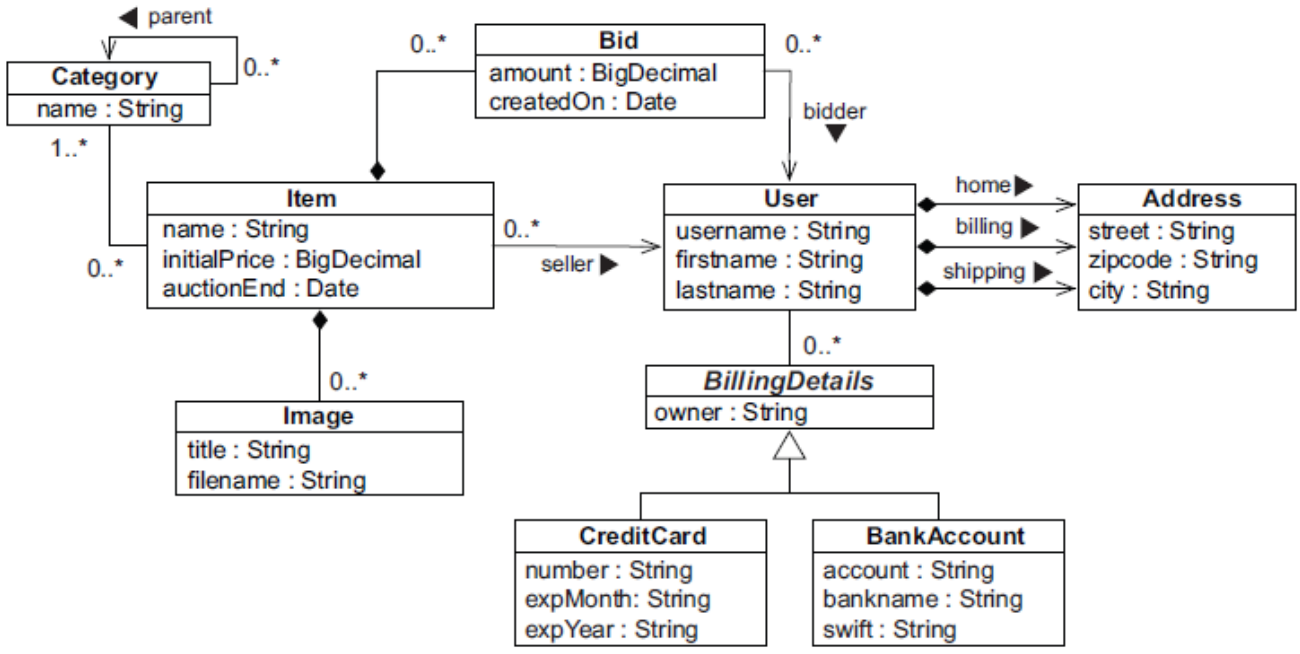
Нативный Hibernate

Пример в JPAHelloWorld. Класс HelloWorldHibernateTest.

Hibernate и JPA

Пример в JPAHelloWorld. Класс HelloWorldSpringDataJPATest.

Доменная модель и метаданные



Поддержка мелкоструктурных доменных моделей - одна из основных задач Hibernate. Это одна из причин, по которой мы работаем с POJO. В целом, использование мелкоструктурных объектов означает, что классов должно быть больше, чем таблиц. В обычном Java-классе с поддержкой персистентности объявляются атрибуты, которые представляют состояние, и бизнес-методы, которые определяют поведение. Некоторые атрибуты представляют собой ассоциации с другими классами,

поддерживающими персистентность. В следующем листинге показана POJO-реализация сущности User доменной модели.

```
public class User {

    private String username;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

}
```

Класс может быть абстрактным и, при необходимости, расширять другой класс или реализовывать интерфейс. Он должен быть классом верхнего уровня, а не вложенным в другой класс. Персистентный класс и любые его методы не должны быть конечными (это требование спецификации JPA). Hibernate не так строг и позволяет объявлять final классы как сущности или сущности с final методами, которые обращаются к полям. Однако это не очень хорошая практика, так как это не позволит Hibernate использовать паттерн прокси для повышения производительности. В целом, если вы хотите, чтобы ваше приложение оставалось переносимым между различными JPA-провайдерами, вам следует следовать требованиям JPA.

Hibernate и JPA требуют наличия конструктора без аргументов для каждого постоянного класса. В качестве альтернативы, если вы вообще не напишете конструктор, Hibernate будет использовать конструктор Java по умолчанию. Hibernate вызывает классы, используя Java Reflection API для таких конструкторов без аргументов, чтобы создать экземпляры. Конструктор не обязательно должен быть общедоступным, но он должен быть, по крайней мере, видимым для пакета, чтобы Hibernate мог использовать генерируемые во время выполнения прокси для оптимизации производительности.

Свойства POJO реализуют атрибуты бизнес-сущностей, например, имя пользователя User. Обычно свойства реализуются в виде приватных или защищенных полей-членов, а также публичных или защищенных методов доступа к свойствам: для каждого поля вам понадобится метод для получения его значения и метод для установки его значения. Эти методы называются getter и setter соответственно.

Спецификация JavaBean определяет правила именования методов-геттеров; это позволяет таким общим инструментам, как Hibernate, легко находить значения свойств и манипулировать ими. Имя метода getter начинается с get, затем следует имя свойства (первая буква в верхнем регистре). Имя метода setter начинается с set и аналогично сопровождается именем свойства. Имена методов getter для булевых свойств можно начинать с is вместо get.

Hibernate не требует методов-акселераторов. Вы можете выбрать, как будет храниться состояние экземпляра ваших классов. Hibernate будет либо напрямую обращаться к полям, либо вызывать методы-аксессоры. Эти соображения не сильно влияют на дизайн вашего класса. Вы можете сделать

некоторые методы-аксессоры непубличными или полностью удалить их, а затем настроить Hibernate на доступ к полям для этих свойств.

Хотя тривиальные методы доступа встречаются часто, одна из причин, по которой нам нравится использовать методы доступа в стиле JavaBeans, заключается в том, что они обеспечивают инкапсуляцию: вы можете изменить скрытую внутреннюю реализацию атрибута, не внося никаких изменений в общедоступный интерфейс. Если вы настроите Hibernate на доступ к атрибутам через методы, вы абстрагируете внутреннюю структуру данных класса - переменные экземпляра - от дизайна базы данных.

Например, если в вашей базе данных имя пользователя хранится в единственном столбце NAME, а в классе User есть поля firstname и lastname, вы можете добавить в класс следующее свойство name:

```
public class User {
    private String firstname;
    private String lastname;

    public String getName() {
        return firstname + ' ' + lastname;
    }

    public void setName(String name) {
        StringTokenizer tokenizer = new StringTokenizer(name);
        firstname = tokenizer.nextToken();
        lastname = tokenizer.nextToken();
    }
}
```

Еще одна проблема, которую следует учитывать, - dirty checking. Hibernate автоматически обнаруживает изменения состояния, чтобы синхронизировать обновленное состояние с базой данных. Обычно безопасно возвращать из метода getter экземпляр, отличный от экземпляра, переданного Hibernate в setter. Hibernate сравнивает их по значению, а не по идентичности объекта, чтобы определить, нужно ли обновлять постоянное состояние атрибута. Например, следующий метод getter не приводит к ненужным SQL UPDATE:

```
public String getFirstname() {
    return new String(firstname);
}
```

Следует отметить важный момент, связанный с dirty checking при сохранении коллекций. Если у вас есть сущность Item с полем Set, доступ к которому осуществляется через сеттер setBids, этот код приведет к ненужному SQL UPDATE:

```
item.setBids(bids);
em.persist(item);
item.setBids(bids);
```

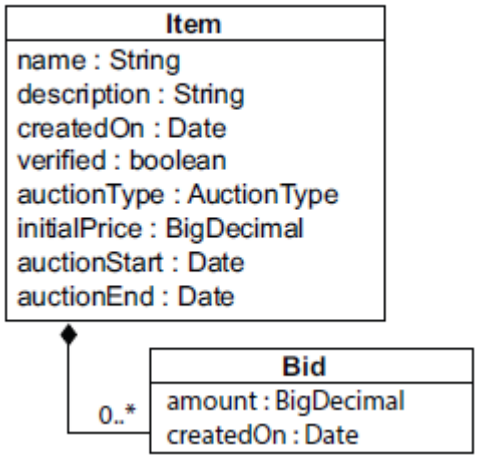
Это происходит потому, что у Hibernate есть свои собственные реализации коллекций: PersistentSet, PersistentList или PersistentMap. Предоставление сеттеров для всей коллекции в любом случае не является хорошей практикой.

Если Hibernate использует методы-аксессоры при загрузке и хранении экземпляров и при этом выбрасывается исключение RuntimeException (без проверки), текущая транзакция откатывается, а исключение обрабатывается в коде, который вызвал Jakarta Persistence (или нативный Hibernate) API. Если вы бросаете проверенное исключение приложения, Hibernate оборачивает его в RuntimeException.

Реализация отношений между POJO

Теперь давайте рассмотрим, как можно связывать и создавать различные виды отношений между объектами: один-ко-многим, многие-к-одному, и двунаправленные отношения.

Вы можете создавать свойства для выражения ассоциаций между классами, и вы будете (обычно) вызывать методы-аксессоры.




```
public class Bid {

    private Item item;

    public Item getItem() {
        return item;
    }

    void setItem(Item item) {
        this.item = item;
    }
}

public class Item {
    private Set<Bid> bids = new HashSet<>();

    public Set<Bid> getBids() {
        return Collections.unmodifiableSet(bids);
```

```
    }

    public void addBid(Bid bid) {
        if (bid == null)
            throw new NullPointerException("Can't add null Bid");
        if (bid.getItem() != null)
            throw new IllegalStateException("Bid is already assigned to an Item");
        bids.add(bid);
        bid.setItem(this);
    }
}
```

Основная процедура связывания Bid с Item выглядит следующим образом:

```
anItem.getBids().add(aBid);
aBid.setItem(anItem);
```

JPA не управляет постоянными ассоциациями. Если вы хотите управлять ассоциацией, вы должны написать тот же код, который вы написали бы без Hibernate. Если ассоциация двунаправленная, вы должны учитывать обе стороны отношений.

Рекомендуется добавить удобные методы для группировки этих операций, что позволит повторно использовать их и обеспечит корректность, а в конечном итоге - целостность данных (Bid обязан иметь ссылку на Item).

Альтернативная стратегия - использовать неизменяемые экземпляры. Например, вы можете обеспечить целостность, требуя аргумент Item в конструкторе Bid.

```
public class Bid {

    private Item item;

    public Bid() {
    }

    public Bid(Item item) {
        this.item = item;
        item.bids.add(this); // Bidirectional
    }

    public Item getItem() {
        return item;
    }

}

public class Item {
    Set<Bid> bids = new HashSet<>();
```

```
    public Set<Bid> getBids() {
        return Collections.unmodifiableSet(bids);
    }

}
```

В этом конструкторе устанавливается поле item; дальнейшее изменение значения поля не требуется. Коллекция на другой стороне также обновляется для двунаправленной связи, а поле bids из класса Item теперь является package-private. Метод Bid#setItem() отсутствует.

Однако с таким подходом есть несколько проблем. Во-первых, Hibernate не может вызвать этот конструктор. Вам нужно добавить конструктор без аргументов для Hibernate, и он должен быть, по крайней мере, package-visible. Кроме того, поскольку метода setItem() не существует, Hibernate придется настраивать на прямой доступ к полю item. Это означает, что поле не может быть конечным, поэтому класс не гарантированно будет неизменяемым.

Метаданные доменной модели

Метаданные - это данные о данных, поэтому метаданные доменной модели - это информация о вашей доменной модели. Например, когда вы используете Java Reflection API, чтобы узнать имена классов в вашей доменной модели или имена их атрибутов, вы обращаетесь к метаданным доменной модели.

Средства ORM также требуют метаданных для определения отображения между классами и таблицами, свойствами и столбцами, ассоциациями и внешними ключами, типами Java и типами SQL и так далее. Эти метаданные объектно-реляционного отображения управляют преобразованием между различными системами типов и представлениями отношений в объектно-ориентированных и SQL-системах. JPA имеет API метаданных, который можно вызвать для получения подробной информации об аспектах персистентности вашей доменной модели, например, имен персистентных сущностей и атрибутов.

JPA стандартизирует два варианта метаданных: аннотации в коде Java и внешние XML-файлы дескрипторов. Hibernate имеет некоторые расширения для собственной функциональности, также доступные в виде аннотаций или XML-дескрипторов. Обычно аннотации предпочтительны в качестве основного источника метаданных отображения.

Большим преимуществом аннотаций является то, что они помещают метаданные, такие как @Entity, рядом с информацией, которую они описывают, вместо того чтобы отделять их в отдельный файл.

Аннотация @Entity отображает определенный класс. JPA и Hibernate также имеют аннотации для глобальных метаданных. Например, аннотация @NamedQuery имеет глобальную область применения; вы не применяете ее к конкретному классу.

Хотя можно поместить такие глобальные аннотации в исходный файл класса (в верхней части любого класса), предпочтительно хранить глобальные метаданные в отдельном файле. Аннотации уровня пакета - хороший выбор; они находятся в файле package-info.java в каталоге определенного пакета. Вы сможете искать их в одном месте, а не просматривать несколько файлов.

Пример в package-info.java

Определение контрактов для объектов

Большинство приложений содержат множество проверок целостности данных. При нарушении одного из простейших ограничений целостности данных вы можете получить исключение NullPointerException, поскольку значение недоступно. Подобное исключение может возникнуть, когда свойство со строковым значением не должно быть пустым (пустая строка не является null), когда строка должна соответствовать определенному шаблону регулярного выражения или когда значение числа или даты должно находиться в определенном диапазоне.

Эти бизнес-правила влияют на каждый уровень приложения: Код пользовательского интерфейса должен отображать подробные и локализованные сообщения об ошибках. Бизнес-слой и слой сохранения должны проверять входные значения, полученные от клиента, прежде чем передавать их в хранилище данных. База данных SQL должна быть окончательным валидатором, гарантирующим целостность долговременных данных.

Идея Bean Validation заключается в том, что объявить такие правила, как «Это свойство не может быть null» или «Это число должно быть в заданном диапазоне», гораздо проще и менее опасно для ошибок, чем многократно писать процедуры if-then-else. Более того, объявление этих правил на центральном компоненте приложения, реализации доменной модели, позволяет проверять целостность на всех уровнях системы. Затем эти правила становятся доступными для уровней представления и сохранения данных. А если учесть, что ограничения целостности данных влияют не только на код вашего Java-приложения, но и на схему базы данных SQL, которая представляет собой набор правил целостности, то ограничения Bean Validation можно рассматривать как дополнительные метаданные ORM.

Метаданные в XML файлах

orm.xml

```
<entity-mappings
        version="2.2"
        xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
            http://xmlns.jcp.org/xml/ns/persistence/orm_2_2.xsd">
    <!--Declare the global metadata.-->
    <persistence-unit-metadata>
        <!--Ignore all mapping annotations. If we include the <xml-mapping-
metadatacomplete>
        element, the JPA provider ignores all annotations on the domain
        model classes in this persistence unit and relies only on the mappings as
defined
        in the XML descriptors.-->
        <xml-mapping-metadata-complete/>
        <!--The default settings escape all SQL columns, tables, and other names.-
->
        <persistence-unit-defaults>
            <!--Escaping is useful if the SQL names are actually keywords (a
"USER" table, for example).-->
            <delimited-identifiers/>
        </persistence-unit-defaults>
```

```
    </persistence-unit-metadata>

    <!--Declare the Item class as an entity with field access.-->
    <entity class="com.sibsutis.study.validation.Item" access="FIELD">
        <!--Its attributes are the id, which is autogenerated, the name, and the
auctionEnd, which is a temporal field.-->
        <attributes>
            <id name="id">
                <generated-value strategy="AUTO"/>
            </id>
            <basic name="name"/>
            <basic name="auctionEnd">
                <temporal>TIMESTAMP</temporal>
            </basic>
        </attributes>
    </entity>

</entity-mappings>
```

Провайдер JPA автоматически подхватывает этот дескриптор, если вы помещаете его в файл META-INF/orm.xml в classpath единицы персистентности. Если вы предпочитаете использовать другое имя файла или несколько файлов, вам придется изменить конфигурацию блока персистентности в файле META-INF/persistence.xml:

```
<persistence-unit name="persistenceUnitName">
. . .
<mapping-file>file1.xml</mapping-file>
<mapping-file>file2.xml</mapping-file>
. . .
</persistence-unit>
```

Если вы не хотите игнорировать метаданные аннотации, а хотите переопределить их, не отмечайте XML-дескрипторы как «полные» и укажите класс и свойство, которые вы хотите переопределить:

```
<entity class="com.manning.javapersistence.ch03.metadataxmljpa.Item">
    <attributes>
        <basic name="name">
            <column name="ITEM_NAME"/>
        </basic>
    </attributes>
</entity>
```

Здесь мы привязываем свойство name к столбцу ITEM_NAME; по умолчанию свойство будет привязано к столбцу NAME. Теперь Hibernate будет игнорировать все существующие аннотации из пакетов jakarta.persistence.annotation и org.hibernate.annotations на свойство name класса Item. Но Hibernate не будет игнорировать аннотации Bean Validation и по-прежнему будет применять их для автоматической валидации и генерации схемы! Все остальные аннотации на классе Item также распознаются. Обратите

внимание, что мы не указываем стратегию доступа в этом отображении, поэтому используется доступ к полю или методы-аксессоры, в зависимости от положения аннотации @Id в Item.

Доступ к метаданным во время исполнения

Спецификация JPA предоставляет программные интерфейсы для доступа к метамодели (информации о модели) персистентных классов. Существует два варианта API. Один из них более динамичен по своей природе и похож на базовую рефлексию Java. Второй вариант - это статическая метамодель. В обоих вариантах доступ предоставляется только для чтения; вы не можете изменять метаданные во время выполнения.

Динамеческое API метамодели

Иногда вам нужен программный доступ к постоянным атрибутам сущности, например, когда вы хотите написать пользовательскую валидацию или общий код пользовательского интерфейса. Вы хотите динамически узнавать, какие персистентные классы и атрибуты есть у вашей доменной модели.

Пример в тестовом методе accessDynamicMetamodel в классе MetamodelTest

Статическое API

В Java (по крайней мере, до версии 17) вы не можете получить доступ к полям или методам-аксессорам бина типобезопасным способом - только по их именам, используя строки. Это особенно неудобно при запросе критериев JPA, который является безопасной с точки зрения типов альтернативой строковым языкам запросов.

Пример в тестовом методе accessStaticMetamodel в классе MetamodelTest

Работа с Spring Data JPA

Spring Data JPA обеспечивает поддержку взаимодействия с JPA репозиториями. Она построена на основе функциональности, предлагаемой проектом Spring Data Commons и JPA-провайдера (в нашем случае Hibernate).

Spring Data JPA может делать несколько вещей для облегчения взаимодействия с базой данных:

- Настроить бин источника данных
- Настройка фабричного бина менеджера сущностей
- Настройка бина менеджера транзакций
- Управление транзакциями с помощью аннотаций

Конфигурация проекта, использующего Spring Data JPA

Пример в проекте springdatajpa

Интерфейс UserRepository расширяет CrudRepository<User, Long>. Это означает, что он является хранилищем сущностей User, которые имеют идентификатор Long. В классе User есть поле id типа Long, аннотированное как @Id. Мы можем напрямую вызывать такие методы, как save, findAll и findById, унаследованные от CrudRepository, и использовать их без дополнительной информации для

выполнения обычных операций с базой данных. Spring Data JPA создаст прокси-класс, реализующий интерфейс UserRepository, и реализует его методы.

Стоит отметить, что CrudRepository - это общий, не зависящий от технологии интерфейс персистентности, который мы можем использовать не только для JPA/реляционных баз данных, но и для NoSQL баз данных. Например, мы можем легко изменить базу данных с PostgreSQL на MongoDB, не трогая реализацию, изменив зависимость с исходного spring-boot-starter-data-jpa на spring-boot-starter-data-mongodb.

Определение методов запросов в Spring Data JPA

Пример в springdatajpa2

Генерация запросов из названий методов:

Ключевое слово	**Пример**	**Сгенерированный JPQL**
Is, Equals	+ findByUsername	. . . where e.username = ?1
	+ findByUsernameIs	
	+ findByUsernameEquals	
And	findByUsernameAndRegistrationDate	. . . where e.username = ?1 and e.registrationdate = ?2
Or	findByUsernameOrRegistrationDate	. . . where e.username = ?1 or e.registrationdate = ?2
LessThan	findByRegistrationDateLessThan	. . . where e.registrationdate < ?1
LessThanEqual	findByRegistrationDateLessThanEqual	. . . where e.registrationdate <= ?1
GreaterThan	findByRegistrationDateGreaterThan	. . . where e.registrationdate > ?1
GreaterThanEqual	findByRegistrationDateGreaterThanEqual	. . . where e.registrationdate >= ?1
Between	findByRegistrationDateBetween	. . . where e.registrationdate between ?1 and ?2
OrderBy	findByRegistrationDateOrderByUsernameDesc	. . . where e. registrationdate = ?1 order by e.username desc
Like	findByUsernameLike . . . where e.username like ?1	
NotLike	findByUsernameNotLike	. . . where e.username not like ?1
Before	findByRegistrationDateBefore	. . . where e.registrationdate < ?1
After	findByRegistrationDateAfter	. . . where e.registrationdate > ?1
Null, IsNull	findByRegistrationDate(Is)Null	. . . where e.registrationdate is null

Ключевое слово	Пример	Сгенерированный JPQL
NotNull, IsNotNull	findByRegistrationDate(Is)NotNull	. . . where e.registrationdate is not null
Not	findByUsernameNot	. . . where e.username <> ?1

Ключевые слова top и first (используемые эквивалентно) могут ограничивать результаты методов запроса. За ключевыми словами top и first может следовать необязательное числовое значение, указывающее на максимальный размер возвращаемого результата. Если это числовое значение отсутствует, то размер результата будет равен 1.

Pageable - это интерфейс для информации о пагинации, но на практике мы используем класс PageRequest, который его реализует. В нем можно указать номер страницы, ее размер и критерий сортировки.

Методы запросов, возвращающие более одного результата, могут использовать стандартные интерфейсы Java, такие как Iterable, List, Set. Кроме того, Spring Data поддерживает Streamable, который можно использовать в качестве альтернативы Iterable или любому типу коллекции. Вы можете объединять Streamable и напрямую фильтровать и отображать элементы.

С помощью аннотации @Query вы можете создать метод, а затем написать к нему пользовательский запрос. При использовании аннотации @Query имя метода не обязательно должно соответствовать какому-либо соглашению об именовании. Пользовательский запрос можно параметризовать, определяя параметры по позиции или по имени и связывая эти имена в запросе с помощью аннотации @Param. Аннотация @Query может генерировать собственные запросы с флагом nativeQuery, установленным в true. Однако следует помнить, что нативные запросы могут повлиять на переносимость приложения. Для сортировки результатов можно использовать объект Sort. Свойства, по которым вы упорядочиваете результаты, должны преобразовываться в свойство запроса или псевдоним запроса. Spring Data JPA поддерживает выражения Spring Expression Language (SpEL) в запросах, определенных с помощью аннотации @Query, а Spring Data JPA поддерживает переменную entityName. В таком запросе, как select e from #{#entityName} e, entityName определяется на основе аннотации @Entity. В нашем случае, в UserRepository extends JpaRepository<User, Long>, entityName разрешится в User.

Проекции

Не все атрибуты сущности всегда нужны, поэтому иногда мы можем обращаться только к некоторым из них. Например, фронтенд может сократить объем ввода-вывода и отображать только ту информацию, которая будет интересна конечному пользователю. Следовательно, вместо возврата экземпляров корневой сущности, управляемой репозиторием, вы можете захотеть создать проекции на основе определенных атрибутов этих сущностей. Spring Data JPA может формировать типы возврата, чтобы выборочно возвращать атрибуты сущностей.

Проекция на основе интерфейса требует создания интерфейса, в котором объявлены методы получения для свойств, включаемых в проекцию. Такой интерфейс также может вычислять конкретные значения с помощью аннотации @Value и выражений SpEL. При выполнении запросов во время

исполнения механизм выполнения создает прокси-объекты интерфейса для каждого возвращаемого элемента и пересылает вызовы открытых методов целевому объекту.

В целом, проекции следует использовать, когда вам нужно предоставить ограниченную информацию и не раскрывать всю сущность. По соображениям производительности следует предпочесть закрытые проекции, когда вы с самого начала знаете, какую информацию хотите вернуть. Если у вас есть запрос, который возвращает полный объект, и аналогичный запрос, который возвращает только проекцию, вы можете использовать альтернативные соглашения об именовании, например, назвать один метод find...By, а другой - get...By.

Проекция на основе класса требует создания класса объекта передачи данных (DTO), в котором объявляются свойства, включаемые в проекцию, и методы геттера. Использование проекции на основе классов аналогично использованию проекций на основе интерфейсов. Однако в Spring Data JPA не нужно создавать прокси-классы для управления проекциями. Spring Data JPA инстанцирует класс, в котором объявлена проекция, а включаемые свойства определяются именами параметров конструктора класса.

Модифицирующие запросы

Вы можете определить модифицирующие методы с помощью аннотации @Modifying. Например, запросы INSERT, UPDATE и DELETE, или операторы DDL, изменяют содержимое базы данных. Аннотация @Query будет содержать модифицирующий запрос в качестве аргумента, и ей могут потребоваться связующие параметры. Такой метод также должен быть аннотирован @Transactional или выполняться из программно управляемой транзакции. Модифицирующие запросы имеют то преимущество, что они четко выделяют, к какому столбцу они обращаются, и они могут включать условия, поэтому они могут сделать код более понятным по сравнению с сохранением или удаления всего объекта. Кроме того, изменение ограниченного числа столбцов в базе данных будет выполняться быстрее. Spring Data JPA также может генерировать запросы на удаление на основе имен методов. Механизм аналогично примерам в таблице, но ключевое слово find заменяется на delete.

Query by example

Query by example (QBE) - это техника составления запросов, которая не требует написания классических запросов, включающих сущности и свойства. Она позволяет динамически создавать запросы и состоит из трех частей: probe, ExampleMatcher и примера. Probe - это объект домена с уже заданными свойствами. ExampleMatcher предоставляет правила для поиска определенных свойств. Пример объединяет probe и ExampleMatcher вместе и генерирует запрос. Несколько примеров могут повторно использовать один ExampleMatcher.

Наиболее подходящие случаи использования для QBE:

- Когда вы отделяете код от базового API хранилища данных.
- Когда происходят частые изменения внутренней структуры доменных объектов, и они не распространяются на существующие запросы.
- При построении набора статических или динамических ограничений для запросов к хранилище.

У QBE есть несколько ограничений:

- Она поддерживает только разные сопоставления для свойств String, и точное соответствие для других типов.
- Она не поддерживает вложенные или сгруппированные фильтры, такие как имя пользователя = ? 0 или (имя пользователя = ?1 и email = ?2).