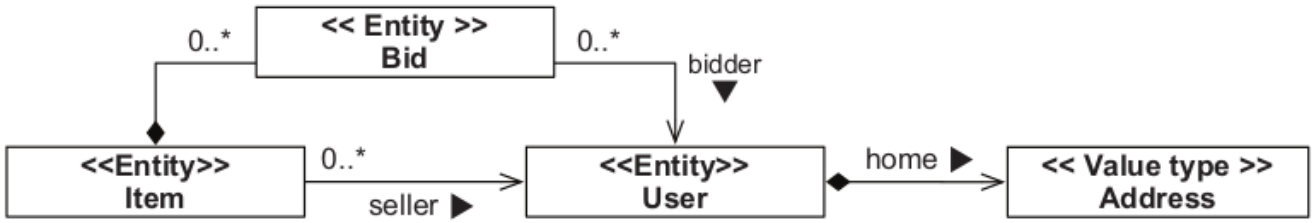
Разработка серверного ПО. Лекция 7. Стратегии отображения объектной модели на реляционную

Основная задача Hibernate и Spring Data JPA, использующих Hibernate в качестве провайдера данных является обеспечение поддержки мелкозернистых (fine-grained) и богатых доменных моделей. Это одна из причин, по которой мы работаем с POJO (Plain Old Java Objects) - обычными Java-объектами, не привязанными к какому-либо фреймворку. В грубом смысле слова «мелкозернистая» модель означает, что классов больше, чем таблиц.

Entity type — можно получить экземпляр типа сущности, используя его идентификатор; например, экземпляр пользователя, элемента или категории. Ссылка на экземпляр сущности (указатель в JVM) сохраняется как ссылка в базе данных (значение, ограниченное внешним ключом). Экземпляр сущности имеет свой собственный жизненный цикл; он может существовать независимо от любой другой сущности. Доменная модель выражается в классах-сущностях.

Value type — экземпляр типа значения не имеет постоянного идентификатора; он принадлежит экземпляру сущности, и его срок жизни связан с принадлежащим ему экземпляром сущности. Экземпляр типа значения не ссылается на другие объекты. В качестве типов значений можно использовать собственные классы доменной модели, например Address и MonetaryAmount.

Если вы прочитаете спецификацию JPA, то обнаружите те же понятия, но типы значений в JPA называются базовыми типами свойств или встраиваемыми классами.



- Общие ссылки — избегайте общих ссылок на экземпляры типов значений при написании классов POJO. Например, убедитесь, что только один пользователь может ссылаться на адрес. Вы можете сделать Address неизменяемым без публичного метода setUser() и обеспечить связь с помощью публичного конструктора с аргументом User.

- Зависимости жизненного цикла — если пользователь удаляется, его зависимость от адреса также должна быть удалена. Метаданные будут включать каскадные правила для всех таких зависимостей, поэтому Hibernate, Spring Data JPA или база данных могут позаботиться об удалении устаревшего адреса. Вы должны спроектировать процедуры и пользовательский интерфейс вашего приложения таким образом, чтобы они учитывали и ожидали таких зависимостей.

- Идентификатор - классы сущностей почти во всех случаях нуждаются в идентификаторе. Value types (и, конечно, классы JDK, такие как String и Integer) не имеют идентификатора,потому что экземпляры идентифицируются через сущность-владельца.

В Java идентичность определяется оператором ==, а равенство методом equals. Персистентность усложняет эту картину. При объектном/реляционном отображении экземпляр сущности - это представление в памяти определенной строки (или строк) таблицы (или таблиц) базы данных. Наряду с идентичностью и равенством в Java мы определяем идентичность базы данных. Теперь у вас есть три способа различать ссылки:

- Объектная идентичность — оператор ==.
- Объектное равенство — метод equals.
- Идентичность базы данных — если объекты, хранящиеся в базе данных, идентичны, то они хранятся в одной таблице и имеют одинаковый первичный ключ.

Выбор первичного ключа

Ключ-кандидат - это столбец или набор столбцов, которые можно использовать для идентификации определенной строки в таблице. Чтобы стать первичным ключом, ключ-кандидат должен удовлетворять следующим требованиям:

- Значение любого столбца ключа-кандидата никогда не бывает null. Вы не можете идентифицировать что-то с помощью неизвестных данных, а в реляционной модели не существует нулей.
- Значение столбца (или столбцов) ключа-кандидата является уникальным значением для любой строки.
- Значение столбца (или столбцов) ключа-кандидата никогда не меняется, оно неизменяемый.

Но Hibernate и Spring Data JPA ожидают, что ключ-кандидат будет неизменяемым, если он используется в качестве первичного ключа. Hibernate и Spring Data JPA с Hibernate в качестве провайдера не поддерживают обновление значений первичных ключей с помощью API

Натуральный первичный ключ — первичный ключ, который имеет бизнес-значение, значим за пределами базы данных. Крайне не рекомендуется выбирать натуральный первичный ключ из-за того, что такие ключи могут изменяться со временем и быть составными, что усложнит поддержку схемы таблицы. Поэтому рекомендуется выбирать суррогатные первичные ключи — специально добавленные столбцы, не имеющие значимости для доменной модели, также их можно автоматически генерировать.

Конфигурация генератора ключей

Аннотация @Id необходима для обозначения свойства идентификатора класса сущности. Без аннотации @GeneratedValue провайдер JPA предполагает, что вы позаботитесь о создании и присвоении значения идентификатора до сохранения экземпляра.

Обычно нужно, чтобы система генерировала значение первичного ключа при сохранении экземпляра сущности, поэтому вы можете написать аннотацию @GeneratedValue рядом с @Id. JPA стандартизирует несколько стратегий генерации значений с помощью перечисления javax.persistence.GenerationType которые вы выбираете с помощью @GeneratedValue(strategy = ...):

- GenerationType.AUTO — Hibernate (или Spring Data JPA, использующий Hibernate в качестве провайдера персистентности) выбирает подходящую стратегию, спрашивая диалект SQL вашей

настроенной базы данных, что лучше. Это эквивалентно @GeneratedValue() без каких-либо настроек.

- GenerationType.SEQUENCE — Hibernate (или Spring Data JPA, использующий Hibernate в качестве провайдера персистентности) ожидает (и создает, если вы используете инструменты) последовательность с именем HIBERNATE_SEQUENCE в вашей базе данных. Эта последовательность будет вызываться отдельно перед каждым INSERT, создавая последовательные числовые значения.
- GenerationType.IDENTITY — Hibernate (или Spring Data JPA, использующий Hibernate в качестве провайдера персистентности) ожидает (и создает в DDL таблицы) специальный автоинкрементный столбец первичного ключа, который автоматически генерирует числовое значение при INSERT в базу данных.
- GenerationType.TABLE — Hibernate (или Spring Data JPA, использующий Hibernate в качестве провайдера персистентности) будет использовать дополнительную таблицу в схеме вашей базы данных, которая содержит следующее числовое значение первичного ключа, с одной строкой для каждого класса сущностей. Эта таблица будет считываться и обновляться перед INSERT. Имя таблицы по умолчанию — HIBERNATE_SEQUENCES с колонками SEQUENCE_NAME и NEXT_VALUE.

В JPA есть две встроенные аннотации, которые можно использовать для настройки именованных генераторов: @javax.persistence.SequenceGenerator и @javax.persistence.TableGenerator.

package-info.java:

```
@org.hibernate.annotations.GenericGenerator(
    name = "ID_GENERATOR",
    strategy = "enhanced-sequence",
    parameters = {
        @org.hibernate.annotations.Parameter(
            name = "sequence_name",
            value = "JPWHSD_SEQUENCE"
        ),
        @org.hibernate.annotations.Parameter(
            name = "initial_value",
            value = "1000"
        )
    })
```

Стратегия enhanced-sequence создает последовательные числовые значения. Если ваш диалект SQL поддерживает последовательности, Hibernate (или Spring Data JPA, использующий Hibernate в качестве провайдера персистентности) будет использовать фактическую последовательность базы данных. Если ваша СУБД не поддерживает собственные последовательности, Hibernate (или Spring Data JPA, использующая Hibernate в качестве провайдера персистентности) будет управлять и использовать дополнительную «таблицу последовательностей», имитируя поведение последовательности.

```
em.getTransaction().begin();
Item item = new Item();
item.setName("Some Item");
item.setAuctionEnd(Helper.tomorrow());
```

```
em.persist(item);
em.getTransaction().commit();
```

В примере выше не задаётся идентификатор, Hibernate сгенерирует его при вставке в базу данных.

В Hibernate существуют следующие стратегии генерации идентификаторов:

- native — эта опция автоматически выбирает стратегию, в зависимости от настроенного диалекта SQL. Это эквивалентно JPA GenerationType.AUTO со старым отображением.
- sequence — Эта стратегия использует собственную последовательность базы данных с именем HIBERNATE_SEQUENCE. Последовательность вызывается перед каждым INSERT новой строки. Вы можете настроить имя последовательности и предоставить дополнительные параметры DDL.
- enhanced-sequence — эта стратегия использует родную последовательность базы данных, если она поддерживается; в противном случае она использует дополнительную таблицу базы данных с одним столбцом и строкой, эмулирующей последовательность (имя таблицы по умолчанию - HIBERNATE_SEQUENCE). Использование этой стратегии всегда вызывает «последовательность» базы данных перед INSERT, обеспечивая одинаковое поведение независимо от того, поддерживает ли СУБД реальные объекты последовательностей. Это лучшая стратегия по умолчанию.
- enhanced-table — эта стратегия использует дополнительную таблицу HIBERNATE_SEQUENCES, в которой по умолчанию одна строка представляет последовательность и хранит следующее значение. Это значение выбирается и обновляется, когда необходимо сгенерировать значение идентификатора.
- identity — эта стратегия поддерживает IDENTITY и автоинкремент в DB2, MySQL, MS SQL Server и Sybase. Значение идентификатора для столбца первичного ключа будет генерироваться при вставке строки. Она не имеет никаких опций. К сожалению, из-за причуды в коде Hibernate вы не можете настроить эту стратегию в @GenericGenerator. Генерация DDL не будет включать опцию IDENTITY или автоинкремент для столбца первичного ключа. Единственный способ использовать её — это использовать JPA GenerationType.IDENTITY
- increment — при запуске Hibernate эта стратегия считывает максимальное (числовое) значение столбца первичного ключа в таблице каждой сущности и увеличивает это значение на единицу при каждой вставке новой строки. Это особенно эффективно, если некластеризованное приложение Hibernate имеет эксклюзивный доступ к базе данных, но не используйте ее в других сценариях.
- select — при этой стратегии Hibernate не будет генерировать значение ключа или включать столбец первичного ключа в оператор INSERT. Hibernate ожидает, что СУБД присвоит значение столбцу при вставке (значение по умолчанию в схеме или значение, заданное триггером). После вставки Hibernate извлекает столбец первичного ключа с помощью запроса SELECT. Необходимым параметром является key, указывающий свойство идентификатора базы данных (например, id) для SELECT. Эта стратегия не очень эффективна и должна использоваться только со старыми драйверами JDBC, которые не могут возвращать сгенерированные ключи напрямую.

- uuid2 — эта стратегия создает уникальный 128-битный UUID на прикладном уровне. Это полезно, когда вам нужны глобально уникальные идентификаторы для всех баз данных (например, если вы объединяете данные из нескольких разных производственных баз данных в архив при пакетном запуске каждую ночь). UUID может быть закодирован либо как java.lang.String, либо как byte[16], либо как свойство java.util.UUID в вашем классе сущности.
- guid — эта стратегия использует глобальный уникальный идентификатор, создаваемый базой данных, с помощью SQL-функции, доступной в Oracle, Ingres, MS SQL Server и MySQL. Hibernate вызывает функцию базы данных перед INSERT. Значение отображается на свойство идентификатора java.lang.String.

Сопоставление имён классов и таблиц

Если вы укажете @Entity для класса, имя сопоставленной таблицы по умолчанию будет таким же, как и имя класса. Например, сущность Item сопоставляется с таблицей ITEM (или item). Сущность с именем BidItem будет сопоставлен с таблицей BID_ITEM (bid_item) (здесь camel-case будет преобразован в snake-case). Можно переопределить имя таблицы с помощью аннотации JPA @Table.

```
@Entity
@Table(name = "USERS")
public class User {
    // . . .
}
```

Hibernate предоставляет возможность автоматически применять стандарты именования.

```
public class CENamingStrategy extends PhysicalNamingStrategyStandardImpl {
    @Override
    public Identifier toPhysicalTableName(Identifier name,
        JdbcEnvironment context) {
        return new Identifier("CE_" + name.getText(), name.isQuoted());
    }
}
```

Используя Hibernate JPA:

```
<persistence-unit name="mapping">
    ...
    <properties>
        ...
        <property name="hibernate.physical_naming_strategy"
        value="ru.sibsutis.study.javapersistence.CENamingStrategy"/>
    </properties>
</persistence-unit>
```

Используя Spring Data JPA с Hibernate в LocalContainerEntityManagerFactoryBean:

```
properties.put("hibernate.physical_naming_strategy",
CENamingStrategy.class.getName());
```

По умолчанию все имена сущностей автоматически импортируются в пространство имен механизма запросов. Другими словами, вы можете использовать короткие имена классов без префикса пакета в строках запросов JPA.

Это работает только в том случае, если у вас один класс Item в блоке персистентности. Если вы добавите еще один класс Item в другой пакет, вам следует переименовать один из них для JPA, если вы хотите продолжать использовать короткую форму в запросах.

```
package my.other.model;
@javax.persistence.Entity(name = "AuctionItem")
public class Item {
// . . .
}
```

По умолчанию Hibernate и Spring Data JPA, использующие Hibernate в качестве провайдера, создают SQL-операторы для каждой сущности при создании единицы персистентности при запуске. Эти операторы представляют собой простые операции создания, чтения, обновления и удаления (CRUD) для чтения одной строки, удаления строки и так далее. Их дешевле создавать и кэшировать вместо того, чтобы генерировать строки SQL каждый раз, когда такой простой запрос должен быть выполнен во время выполнения. Кроме того, кэширование подготовленных операторов на уровне JDBC гораздо эффективнее если операторов меньше.

Генерируемый SQL запрос обновляет все столбцы, и если значение конкретного столбца не изменяется, то оператор устанавливает его старое значение.

В некоторых ситуациях, таких как, таблицы с сотнями столбцов, необходимо отключить генерацию запросов. Для того чтобы это сделать нужно пометить сущности для которых это делается аннотациями:

```
@Entity
@org.hibernate.annotations.DynamicInsert
@org.hibernate.annotations.DynamicUpdate
public class Item {
// . . .
}
```

После этого вставка и модификация будет происходить только с указанными в запросе столбцами.

Если нужно запретить изменять сущность, то её необходимо пометить аннотацией

```
@Entity
@org.hibernate.annotations.Immutable
public class Bid {
// . . .
}
```

Создание представления на уровне приложения:

```
@Entity
@org.hibernate.annotations.Immutable
@org.hibernate.annotations.Subselect(
    value = "select i.ID as ITEMID, i.NAME as NAME, " +
    "count(b.ID) as NUMBEROFBIDS " +
    "from ITEM i left outer join BID b on i.ID = b.ITEM_ID " +
    "group by i.ID, i.NAME"
)
@org.hibernate.annotations.Synchronize({"ITEM", "BID"})
public class ItemBidSummary {
    @Id
    private Long itemId;
    private String name;
    private long numberOfBids;
    public ItemBidSummary() {
    }
    // Getter methods . . .
    // . . .
}
```

В аннотации @org.hibernate.annotations.Synchronize следует перечислить все имена таблиц, на которые ссылается ваш SELECT. Тогда фреймворк будет знать, что ему нужно синхронизировать Item и Bid перед выполнением запроса к ItemBidSummary. Если в памяти есть изменения, которые еще не были сохранены в базе данных, но могут повлиять на запрос, Hibernate (или Spring Data JPA, использующий Hibernate в качестве провайдера) обнаружит это и примнит изменения перед выполнением запроса. В противном случае результатом может стать неактуальное состояние. Поскольку на классе ItemBidSummary нет аннотации @Table, фреймворк не знает, когда он должен выполнить автоматическую синхронизацию. Аннотация @org.hibernate.annotations.Synchronize указывает, что фреймворк должен синхронизировать таблицы ITEM и BID перед выполнением запроса.

Использование класса сущности ItemBidSummary из Hibernate JPA, доступного только для чтения, будет выглядеть следующим образом:

```
TypedQuery<ItemBidSummary> query = em.createQuery("select ibs from ItemBidSummary
ibs where ibs.itemId = :id", ItemBidSummary.class);
ItemBidSummary itemBidSummary = query.setParameter("id", 1000L).getSingleResult();
```

Или через репозиторий:

```
public interface ItemBidSummaryRepository extends CrudRepository<ItemBidSummary,
Long> {
}

Optional<ItemBidSummary> itemBidSummary =
itemBidSummaryRepository.findById(1000L);
```

Отображение Value types

Отображение базовых типов

- Базовыми типами будем называть примитивные типы Java, классы-обёртки над примитивами и классы String, BigInteger, BigDecimal, java.time.LocalDateTime, java.time.LocalDate, java.time.LocalDate, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, byte[], Byte[], char[], Character[]. Hibernate или Spring Data JPA с помощью Hibernate загружает и сохраняет значение свойства в столбце с соответствующим типом SQL и тем же именем, что и свойство.
- В противном случае, если мы аннотируем класс свойства как @Embeddable или обозначим само свойство как @Embedded, свойство будет отображено как встроенный компонент класса-владельца.
- В противном случае, если тип свойства — java.io.Serializable, его значение будет храниться в сериализованном виде. Это может вызвать проблемы совместимости (мы могли хранить информацию в одном формате класса, а позже захотим получить ее в другом формате) и проблемы производительности (операции сериализации/десериализации требуют больших затрат). Мы всегда должны отображать классы Java, а не хранить в базе данных серию байтов. Сохранение в базе данных двоичной информации, когда приложение может исчезнуть через несколько лет, будет означать, что классы, к которым привязана сериализованная версия, больше недоступны.

Такое поведение означает, что нет необходимости специально отображать классы на таблицы, но делать это в исключительных случаях.

Возможно, мы не хотим, чтобы все свойства класса сущности отображались на столбцы таблицы. Например, хотя имеет смысл иметь свойство Item#initialPrice, свойство Item#totalPriceIncludingTax не должно сохраняться в базе данных, если мы вычисляем и используем его значение только во время выполнения. Чтобы исключить свойство, пометьте поле или метод геттера свойства аннотацией @javax.persistence.Transient или используйте ключевое слово transient. Ключевое слово transient исключает поля как при сериализации в Java, так и при сохранении, поскольку оно также распознается провайдерами JPA. Аннотация @javax.persistence.Transient только исключит поле из отображения.

Если мы не хотим полагаться на стандартные значения отображения свойств, мы можем применить аннотацию @Basic к определенному свойству. @Basic имеет только два свойства optional и fetch. Если нужно указать, что свойство не должно быть null, то вместо @Basic следует использовать @Column.

@Column позволяет контролировать некоторые параметры SQL, например, catalog и schema.

Настройка доступа к свойствам

Механизм персистентности получает доступ к свойствам класса либо напрямую через поля, либо косвенно через геттеры и сеттеры. Сейчас мы попытаемся ответить на вопрос: «Как мы должны обращаться к каждому персистентному свойству?» Аннотированная сущность наследует работу со свойствами от обязательной аннотации @Id. Например, если мы объявляем @Id для поля, а не используем геттер, то все остальные аннотации отображения для этой сущности должны быть полями. Аннотации не поддерживаются для сеттеров.

Стратегия доступа по умолчанию применима не только к одному классу-сущности. Любой @Embedded класс наследует стратегию доступа по умолчанию или явно объявленную стратегию доступа принадлежащего ему корневого класса-сущности.

Спецификация JPA предлагает аннотацию @Access для переопределения поведения по умолчанию, используя параметры AccessType.FIELD (доступ через поля) и AccessType.PROPERTY (доступ через геттеры). Когда вы устанавливаете @Access на уровне класса или сущности, доступ ко всем свойствам класса будет осуществляться в соответствии с выбранной стратегией. Любые другие аннотации отображения, включая @Id, могут быть установлены как на поля, так и на геттеры.

Мы также можем использовать аннотацию @Access, чтобы переопределить стратегию доступа к отдельным свойствам, как в следующем примере. Обратите внимание, что положение других аннотаций отображения, например @Column, не меняется - меняется только способ доступа к экземплярам во время выполнения.

```
@Entity
public class Item {
    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    private Long id;

    @Access(AccessType.PROPERTY)
    @Column(name = "ITEM_NAME")
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = !name.startsWith("AUCTION: ") ? "AUCTION: " + name : name;
    }
}
```

Значение производного свойства вычисляется во время выполнения программы путем вычисления SQL-выражения, объявленного с помощью аннотации @org.hibernate.annotations.Formula.

```
@Formula(
    "CONCAT(SUBSTR(DESCRIPTION, 1, 12), '...')"
```

```
)
private String shortDescription;

@Formula(
    "(SELECT AVG(B.AMOUNT) FROM BID B WHERE B.ITEM_ID = ID)"
)
private BigDecimal averageBidAmount;
```

Формулы SQL вычисляются каждый раз, когда сущность Item извлекается из базы данных, а не в другое время, поэтому результат может устареть, если другие свойства будут изменены. Свойства никогда не появляются в SQL INSERT или UPDATE, только в SELECT. Вычисление происходит в базе данных; формула SQL встраивается в предложение SELECT при загрузке экземпляра.

Трансформация значений столбцов

Теперь давайте разберемся с информацией, которая имеет разное представление в объектно-ориентированной и реляционной системах. Предположим, в базе данных есть столбец IMPERIALWEIGHT, хранящий вес элемента в фунтах. Однако в приложении свойство Item#metricWeight хранится в килограммах, поэтому нам придется преобразовывать значение столбца базы данных при чтении строки из таблицы ITEM и записи ее в таблицу ITEM. Мы можем реализовать это с помощью расширения Hibernate: аннотации @org.hibernate.annotations.ColumnTransformer.

```
@Column(name = "IMPERIALWEIGHT")
@ColumnTransformer(read = "IMPERIALWEIGHT / 2.20462", write = "? * 2.20462")
private double metricWeight;
```

```
List<Item> result = em.createQuery("SELECT i FROM Item i WHERE i.metricWeight =
:w")
.setParameter("w", 2.0)
.getResultList();
```

Сгенерированные значения и значения по умолчанию

Обычно приложениям Hibernate (или Spring Data JPA, использующим Hibernate) требуется обновлять экземпляры, содержащие свойства, для которых база данных генерирует значения после сохранения. Это означает, что при вставке или обновлении строки приложению придется еще раз обращаться к базе данных, чтобы считать значение. Однако пометка свойств как генерируемых позволяет приложению делегировать эту ответственность Hibernate или Spring Data JPA, использующему Hibernate. По сути, каждый раз, когда SQL INSERT или UPDATE выполняется для сущности, объявившей генерируемые свойства, SQL сразу после этого выполняет SELECT для получения генерируемых значений.

Мы используем аннотацию @org.hibernate.annotations.Generated, чтобы отметить сгенерированные свойства. Для временных свойств мы используем аннотации @CreationTimestamp и @UpdateTimestamp. Аннотация @CreationTimestamp используется для пометки свойства createdOn. Это указывает Hibernate

или Spring Data, использующим Hibernate, на автоматическую генерацию значения свойства. В данном случае значение устанавливается на текущую дату перед вставкой экземпляра сущности в базу данных. Другой похожей встроенной аннотацией является @UpdateTimestamp, которая автоматически генерирует значение свойства при обновлении экземпляра сущности.

```
@CreationTimestamp
private LocalDate createdOn;

@UpdateTimestamp
private LocalDateTime lastModified;

@Column(insertable = false)
@ColumnDefault("1.00")
@Generated(
    org.hibernate.annotations.GenerationTime.INSERT
)
private BigDecimal initialPrice;
```

Для перечисления GenerationTime доступны следующие настройки: ALWAYS и INSERT. При использовании GenerationTime.ALWAYS Hibernate или Spring Data JPA, использующий Hibernate, обновляет экземпляр сущности после каждого SQL UPDATE или INSERT. При использовании GenerationTime.INSERT обновление происходит только после SQL INSERT для получения значения по умолчанию, предоставленного базой данных. Мы также можем отобразить свойство initialPrice как не вставляемое. Аннотация @ColumnDefault устанавливает значение по умолчанию для столбца, когда Hibernate или Spring Data JPA, использующая Hibernate, экспортирует и генерирует DDL схемы SQL.

Временные метки часто автоматически генерируются либо базой данных, как в предыдущем примере, либо приложением. Пока мы используем JPA 2.2 и классы Java 8 LocalDate, LocalDateTime и LocalTime, нам не нужно использовать аннотацию @Temporal. Перечисляемые классы Java 8 из пакета java.time сами по себе включают настройки временной точности: дату, дату и время или только время.

Аннотация @Temporal

Аннотация @Temporal позволяет указать точный тип данных SQL при отображении.

```
@CreationTimestamp
@Temporal(TemporalType.DATE)
private Date createdOn;

@UpdateTimestamp
@Temporal(TemporalType.TIMESTAMP)
private Date lastModified;
```

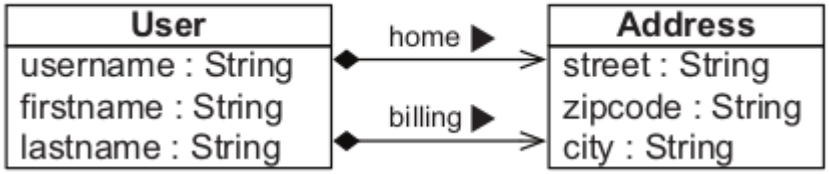
Отображение перечислений

```
```

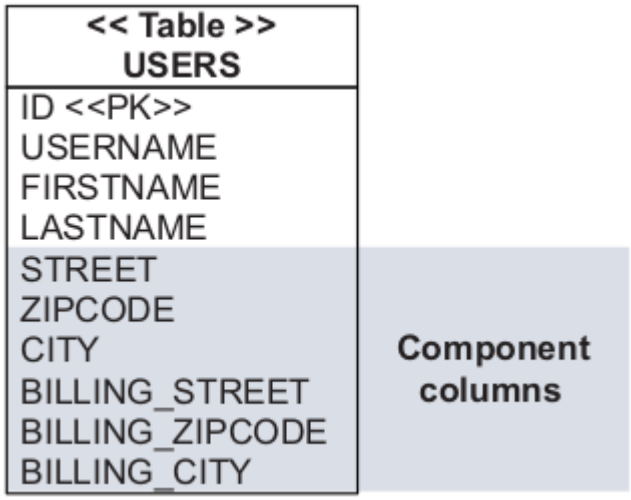
```
@NotNull
@Enumerated(EnumType.STRING)
private AuctionType auctionType = AuctionType.HIGHEST_BID;
```

Без аннотации @Enumerated Hibernate или Spring Data JPA, использующий Hibernate будет хранить ОРДИНАЛЬНУЮ позицию значения. То есть будет храниться 1 для HIGHEST_BID, 2 для LOWEST_BID и 3 для FIXED_PRICE. Это хрупкое значение по умолчанию; если вы внесете изменения в перечисление AuctionType и добавите новый экземпляр, существующие значения могут перестать соответствовать той же позиции и нарушить работу приложения. Поэтому вариант EnumType.STRING является лучшим выбором; Hibernate или Spring Data JPA, использующий Hibernate, может хранить метку значения перечисления как есть.

Отображение встраиваемых компонентов



Объект Address не может существовать в отсутствие объекта User, поэтому составной класс в UML, такой как Address, часто является кандидатом на тип значения для объектно-реляционного отображения.




```
@Embeddable
public class Address {
    @NotNull
    @Column(nullable = false)
    private String street;
    @NotNull
    @Column(nullable = false, length = 5)
    private String zipcode;
    @NotNull
```

```
    @Column(nullable = false)
    private String city;
    public Address() {
    }
    public Address(String street, String zipcode, String city) {
    this.street = street;
    this.zipcode = zipcode;
    this.city = city;
    }
    //getters and setters
}

@Entity
@Table(name = "USERS")
public class User {
    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    private Long id;
    private Address homeAddress;
    // . . .
}
```

```
@Entity
@Table(name = "USERS")
public class User {
    @Embedded
    @AttributeOverride(name = "street",
        column = @Column(name = "BILLING_STREET"))
    @AttributeOverride(name = "zipcode",
        column = @Column(name = "BILLING_ZIPCODE", length = 5))
    @AttributeOverride(name = "city",
        column = @Column(name = "BILLING_CITY"))
    private Address billingAddress;

    public Address getBillingAddress() {
        return billingAddress;
    }
    public void setBillingAddress(Address billingAddress) {
        this.billingAddress = billingAddress;
    }
    // . . .
}
```

@AttributeOverride выборочно отображает атрибуты на встроенный тип.

Репозиторий:

```
public interface UserRepository extends CrudRepository<User, Long> {
}
```

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {SpringDataConfiguration.class})
public class MappingValuesSpringDataJPATest {
    @Autowired
    private UserRepository userRepository;

    @Autowired
    private ItemRepository itemRepository;

    @Test
    void storeLoadEntities() {
        User user = new User();
        user.setUsername("username");
        user.setHomeAddress(new Address("Flowers Street",
            "12345", "Boston"));

        userRepository.save(user);

        Item item = new Item();
        item.setName("Some Item");
        item.setMetricWeight(2);
        item.setDescription("descriptiondescription");

        itemRepository.save(item);

        List<User> users = (List<User>) userRepository.findAll();
        List<Item> items = (List<Item>)

        itemRepository.findByMetricWeight(2.0);
        assertAll(
            () -> assertEquals(1, users.size()),
            () -> assertEquals("username", users.get(0).getUsername()),
            () -> assertEquals("Flowers Street",
                users.get(0).getHomeAddress().getStreet()),
            () -> assertEquals("12345",
                users.get(0).getHomeAddress().getZipcode()),
            () -> assertEquals("Boston",
                users.get(0).getHomeAddress().getCity()),
            () -> assertEquals(1, items.size()),
            () -> assertEquals("AUCTION: Some Item", items.get(0).getName()),
            () -> assertEquals("descriptiondescription",
                items.get(0).getDescription()),
            () -> assertEquals(AuctionType.HIGHEST_BID,
                items.get(0).getAuctionType()),
            () -> assertEquals("descriptiond...",
                items.get(0).getShortDescription()),
            () -> assertEquals(2.0, items.get(0).getMetricWeight()),
            () -> assertEquals(LocalDate.now(),
                items.get(0).getCreatedOn()),
            () ->
            assertTrue(ChronoUnit.SECONDS.between(
```

```
                LocalDateTime.now(),
                items.get(0).getLastModified()) < 1),
            () -> assertEquals(new BigDecimal("1.00"),
                items.get(0).getInitialPrice()));
    }
}
```

Вложенные встраиваемые компоненты

```
@Embeddable
public class Address {
    @NotNull
    @Column(nullable = false)
    private String street;

    @NotNull
    @AttributeOverride(
        name = "name",
        column = @Column(name = "CITY", nullable = false)
    )

    private City city;
    // . . .
}

@Embeddable
public class City {
    @NotNull
    @Column(nullable = false, length = 5)
    private String zipcode;

    @NotNull
    @Column(nullable = false)
    private String name;

    @NotNull
    @Column(nullable = false)
    private String country;
    // . . .
}
```

Отображение примитивных типов

Название типа	Java тип	ANSI SQL тип
integer	int, java.lang.Integer	INTEGER
long	long, java.lang.Long	BIGINT
short	short, java.lang.Short	SMALLINT

Название типа	Java тип	ANSI SQL тип
float	float, java.lang.Float	FLOAT
double	double, java.lang.Double	DOUBLE
byte	byte, java.lang.Byte	TINYINT
boolean	boolean, java.lang.Boolean	BOOLEAN
big_decimal	java.math.BigDecimal	NUMERIC
big_integer	java.math.BigInteger	NUMERIC

Отображение символьных типов

Название типа	Java тип	ANSI SQL тип
string	java.lang.String	VARCHAR
character	char[], Character[], java.lang.String	CHAR
yes_no	boolean, java.lang.Boolean	CHAR(1), 'Y' or 'N'
true_false	boolean, java.lang.Boolean	CHAR(1), 'T' or 'F'
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

Отображение даты и времени

Название типа	Java тип	ANSI SQL тип
date	java.util.Date, java.sql.Date	DATE
time	java.util.Date, java.sql.Time	TIME
timestamp	java.util.Date, java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE
duration	java.time.Duration	BIGINT
instant	java.time.Instant	TIMESTAMP
localdatetime	java.time.LocalDateTime	TIMESTAMP
localdate	java.time.LocalDate	DATE
localtime	java.time.LocalTime	TIME

Название типа	Java тип	ANSI SQL тип
offsetdatetime	java.time.OffsetDateTime	TIMESTAMP
offsettime	java.time.OffsetTime	TIME
zoneddatetime	java.time.ZonedDateTime	TIMESTAMP

Отображение типов больших объёмов

Название типа	Java тип	ANSI SQL тип
binary	byte[], java.lang.Byte[]	VARBINARY
text	java.lang.String	CLOB
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB
serializable	java.io.Serializable	VARBINARY

```
@Entity
public class Item {
    @Lob
    private byte[] image;
    @Lob
    private String description;
}
```

```
Session session = em.unwrap(Session.class);

Blob blob = session.getLobHelper()
    .createBlob(imageInputStream, byteLength);

someItem.setImageBlob(blob);
em.persist(someItem);
```

Выбор адаптера типа происходит при помощи аннотации @Type:

```
@Entity
public class Item {
    @org.hibernate.annotations.Type(type = "yes_no")
    private boolean verified = false;
}
```

Создание собственных адаптеров типов

```
public class MonetaryAmount implements Serializable {
    private final BigDecimal value;
    private final Currency currency;

    public MonetaryAmount(BigDecimal value, Currency currency) {
        this.value = value;
        this.currency = currency;
    }

    public BigDecimal getValue() {
        return value;
    }

    public Currency getCurrency() {
        return currency;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        MonetaryAmount that = (MonetaryAmount) o;
        return Objects.equals(value, that.value) &&
        Objects.equals(currency, that.currency);
    }

    public int hashCode() {
        return Objects.hash(value, currency);
    }

    public String toString() {
        return value + " " + currency;
    }

    public static MonetaryAmount fromString(String s) {
        String[] split = s.split(" ");
        return new MonetaryAmount(
            new BigDecimal(split[0]),
            Currency.getInstance(split[1])
        );
    }
}
```

```
@Converter
public class MonetaryAmountConverter
implements AttributeConverter<MonetaryAmount, String> {

    @Override
    public String convertToDatabaseColumn(MonetaryAmount monetaryAmount) {
        return monetaryAmount.toString();
```

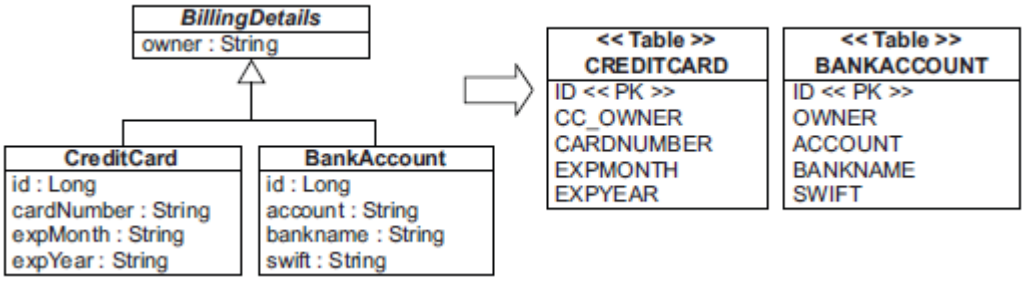
```
    }

    @Override
    public MonetaryAmount convertToEntityAttribute(String s) {
        return MonetaryAmount.fromString(s);
    }
}
```

Отображение наследования

Существует четыре различных стратегии представления иерархии наследования:

- Использовать одну таблицу на конкретный класс и полиморфное поведение по умолчанию во время выполнения.
- Использовать одну таблицу на конкретный класс, но полностью исключить полиморфизм и отношения наследования из схемы SQL. Использовать UNION SQL-запросы для полиморфного поведения во время выполнения.
- Использовать одну таблицу на иерархию классов: отобразить полиморфизм, денормализовав схему SQL.
- Использовать одну таблицу для каждого подкласса: представить отношения is-a (наследование) как отношения has-a (внешний ключ) и использовать JOIN.

Таблица для конкретного класса (отображение с неявным полиморфизмом)




```
@MappedSuperclass
public abstract class BillingDetails {
    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    private Long id;
    @NotNull
    private String owner;
    // . . .
}

@Entity
@AttributeOverride(
name = "owner",
column = @Column(name = "CC_OWNER", nullable = false))
public class CreditCard extends BillingDetails {
    @NotNull
```

```
    private String cardNumber;
    @NotNull
    private String expMonth;
    @NotNull
    private String expYear;
    // . . .
}

@Entity
public class BankAccount extends BillingDetails {
    @NotNull
    private String account;
    @NotNull
    private String bankname;
    @NotNull
    private String swift;
    // . . .
}
```

Репозитории:

```
@NoRepositoryBean
public interface BillingDetailsRepository<T extends BillingDetails, ID>
extends JpaRepository<T, ID> {
    List<T> findByOwner(String owner);
}

public interface BankAccountRepository
extends BillingDetailsRepository<BankAccount, Long> {
    List<BankAccount> findBySwift(String swift);
}

public interface CreditCardRepository
extends BillingDetailsRepository<CreditCard, Long> {
    List<CreditCard> findByExpYear(String expYear);
}
```

@NoRepositoryBean, чтобы репозиторий не инстанцировался, потому что таблицы BillingDetails не существует.

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {SpringDataConfiguration.class})
public class MappingInheritanceSpringDataJPATest {
    @Autowired
    private CreditCardRepository crediCardRepository;

    @Autowired
    private BankAccountRepository bankAccountRepository;
```

```
    @Test
    void storeLoadEntities() {
        CreditCard creditCard = new CreditCard(
        "John Smith", "123456789", "10", "2030");
        creditCardRepository.save(creditCard);

        BankAccount bankAccount = new BankAccount(
        "Mike Johnson", "12345", "Delta Bank", "BANKXY12");
        bankAccountRepository.save(bankAccount);

        List<CreditCard> creditCards =
        creditCardRepository.findByOwner("John Smith");

        List<BankAccount> bankAccounts =
        bankAccountRepository.findByOwner("Mike Johnson");

        List<CreditCard> creditCards2 =
        creditCardRepository.findByExpYear("2030");

        List<BankAccount> bankAccounts2 =
        bankAccountRepository.findBySwift("BANKXY12");

        assertAll(
        () -> assertEquals(1, creditCards.size()),
        () -> assertEquals("123456789",
        creditCards.get(0).getCardNumber()),
        () -> assertEquals(1, bankAccounts.size()),
        () -> assertEquals("12345",
        bankAccounts.get(0).getAccount()),
        () -> assertEquals(1, creditCards2.size()),
        () -> assertEquals("John Smith",
        creditCards2.get(0).getOwner()),
        () -> assertEquals(1, bankAccounts2.size()),
        () -> assertEquals("Mike Johnson",
        bankAccounts2.get(0).getOwner()));
    }
}
```

Основная проблема с неявным отображением наследования заключается в том, что оно не очень хорошо поддерживает полиморфные ассоциации. В базе данных мы обычно представляем ассоциации как отношения внешних ключей.

Если все подклассы отображены на разные таблицы, полиморфная ассоциация с их суперклассом (абстрактным BillingDetails) не может быть представлена в виде простого отношения внешнего ключа. У нас не может быть другой сущности, сопоставленной с внешним ключом «ссылающейся на BILLINGDETAILS» - такой таблицы не существует. Это было бы проблематично в доменной модели, потому что BillingDetails связана с User; обе таблицы CREDITCARD и BANKACCOUNT нуждаются в ссылке внешнего ключа на таблицу USERS. Ни одна из этих проблем не может быть легко решена.

Полиморфные запросы, возвращающие экземпляры всех классов, которые соответствуют интерфейсу запрашиваемого класса, также проблематичны. Hibernate должен выполнить запрос к суперклассу в

виде нескольких SQL SELECT - по одному для каждого конкретного подкласса. Запрос JPA select bd from BillingDetails bd требует двух операторов SQL:

```
select
    ID, OWNER, ACCOUNT, BANKNAME, SWIFT
from
    BANKACCOUNT
select
    ID, CC_OWNER, CARDNUMBER, EXPMONTH, EXPYEAR
from
    CREDITCARD
```

Hibernate или Spring Data JPA, использующий Hibernate, использует отдельный SQL-запрос для каждого конкретного подкласса. С другой стороны, запросы к конкретным классам тривиальны и хорошо выполняются - Hibernate использует только один из запросов.

Еще одна концептуальная проблема этой стратегии отображения заключается в том, что несколько различных столбцов разных таблиц имеют совершенно одинаковую семантику. Это делает эволюцию схемы более сложной. Например, переименование или изменение типа свойства суперкласса приводит к изменению нескольких столбцов в нескольких таблицах. Гораздо сложнее реализовать ограничения целостности базы данных, которые применяются ко всем подклассам.

Таблица на класс с операцией объединением

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {
    @Id
    @GeneratedValue(generator = “ID_GENERATOR”)
    private Long id;
    @NotNull
    private String owner;
    // . . .
}

@Entity
@AttributeOverride(
name = "owner",
column = @Column(name = "CC_OWNER", nullable = false))
public class CreditCard extends BillingDetails {
    @NotNull
    private String cardNumber;
    @NotNull
    private String expMonth;
    @NotNull
    private String expYear;
    // . . .
}
```

```
@Entity
public class BankAccount extends BillingDetails {
    @NotNull
    private String account;
    @NotNull
    private String bankName;
    @NotNull
    private String swift;
    // . . .
}
```

Репозиторий:

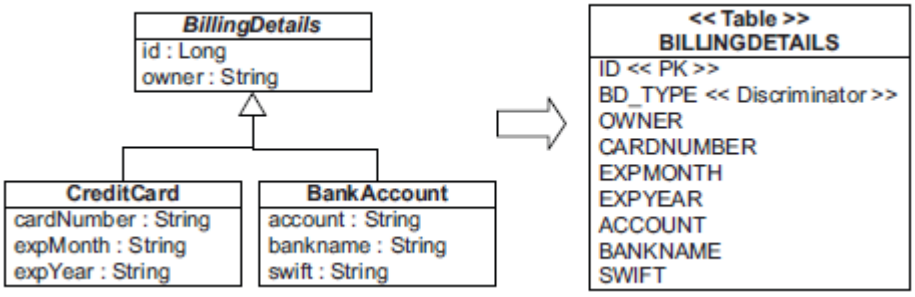
```
public interface BillingDetailsRepository<T extends BillingDetails, ID>
extends JpaRepository<T, ID> {
    List<T> findByOwner(String owner);
}
```

```
billingDetailsRepository.findAll();
```

сгенерирует следующий SQL код:

```
select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER,
    ACCOUNT, BANKNAME, SWIFT, CLAZZ_
from
    ( select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER,
    null as ACCOUNT,
    null as BANKNAME,
    null as SWIFT,
    1 as CLAZZ_
    from
    CREDITCARD
    union all
    select
    id, OWNER,
    null as EXPMONTH,
    null as EXPYEAR,
    null as CARDNUMBER,
    ACCOUNT, BANKNAME, SWIFT,
    2 as CLAZZ_
    from
    BANKACCOUNT
) as BILLINGDETAILS
```

Таблица на иерархию



Эта стратегия отображения является выигрышной с точки зрения производительности и простоты. Это самый эффективный способ представления полиморфизма - как полиморфные, так и неполиморфные запросы работают хорошо, и их даже легко писать вручную. Можно создавать специальные отчеты без сложных объединений или соединений. Эволюция схемы проста. Есть одна серьезная проблема: целостность данных. Мы должны объявить столбцы для свойств, объявленных подклассами, нулевыми. Если в каждом подклассе определено несколько ненулевых свойств, то потеря ограничений NOT NULL может стать серьезной проблемой с точки зрения корректности данных. Представьте, что требуется указать срок действия кредитной карты, но схема базы данных не может обеспечить выполнение этого правила, поскольку все столбцы таблицы могут быть NULL. Простая ошибка прикладного программирования может привести к недействительным данным. Еще одна важная проблема - нормализация. Мы создали функциональные зависимости между столбцами без ключей, нарушив третью нормальную форму. Как всегда, денормализация по соображениям производительности может ввести в заблуждение, поскольку она жертвует долгосрочной стабильностью, удобством обслуживания и целостностью данных ради сиюминутной выгоды, которая может быть достигнута за счет надлежащей оптимизации планов выполнения SQL.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "BD_TYPE")
public abstract class BillingDetails {
    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    private Long id;
    @NotNull
    @Column(nullable = false)
    private String owner;
    // . . .
}
```

Корневой класс иерархии наследования, BillingDetails, автоматически сопоставляется с таблицей BILLINGDETAILS. Общие свойства суперкласса могут быть NOT NULL в схеме; каждый экземпляр подкласса должен иметь значение. Причуда реализации Hibernate требует, чтобы мы объявили нулевость с помощью @Column, потому что Hibernate игнорирует @NotNull в Bean Validation, когда генерирует схему базы данных.

Мы должны добавить специальный столбец-дискриминатор, чтобы различать, что представляет собой каждая строка. Это не свойство сущности; оно используется Hibernate внутри. Имя столбца - BD_TYPE, а

значения - строки, в данном случае «CC» или «BA». Hibernate или Spring Data JPA, использующий Hibernate, автоматически устанавливает и извлекает значения дискриминатора.

Если мы не указываем столбец дискриминатора в суперклассе, его имя по умолчанию принимает значение DTYPE, а значения являются строками. Все конкретные классы в иерархии наследования могут иметь значение дискриминатора, например CreditCard.

```
@Entity
@DiscriminatorValue("CC")
public class CreditCard extends BillingDetails {
    @NotNull
    private String cardNumber;
    @NotNull
    private String expMonth;
    @NotNull
    private String expYear;
    // . . .
}
```

Вызовы

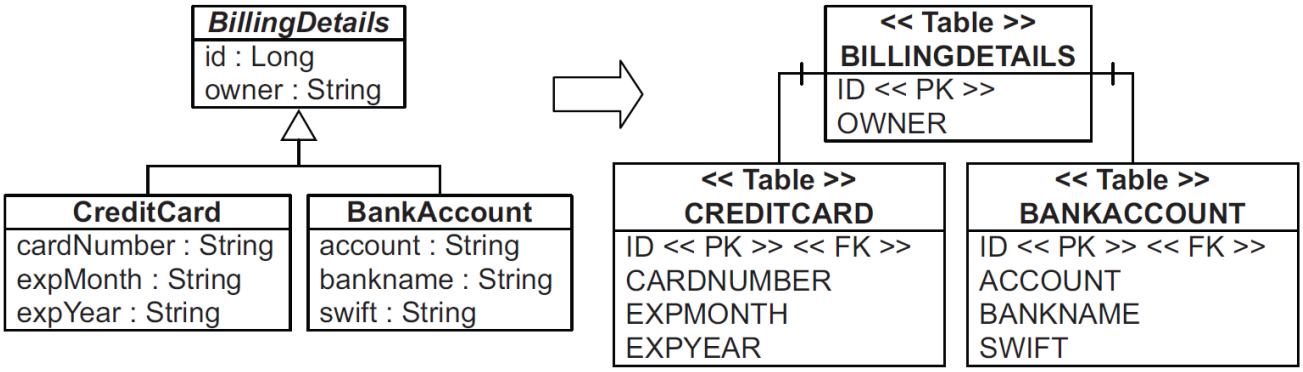
```
billingDetailsRepository.findAll();
creditCardRepository.findAll();
```

Сгенерируют:

```
select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER,
    ACCOUNT, BANKNAME, SWIFT, BD_TYPE
from
    BILLINGDETAILS

select
    ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER
from
    BILLINGDETAILS
where
    BD_TYPE='CC'
```

Соответственно.

Таблица на подкласс с JOIN




```java
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails {
    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    private Long id;
    @NotNull
    private String owner;
    // . . .
}

@Entity
public class BankAccount extends BillingDetails {
    @NotNull
    private String account;
    @NotNull
    private String bankname;
    @NotNull
    private String swift;
    // . . .
}

@Entity
@PrimaryKeyJoinColumn(name = "CREDITCARD_ID")
public class CreditCard extends BillingDetails {
    @NotNull
    private String cardNumber;
    @NotNull
    private String expMonth;
    @NotNull
    private String expYear;
    // . . .
}
```

Вызовы

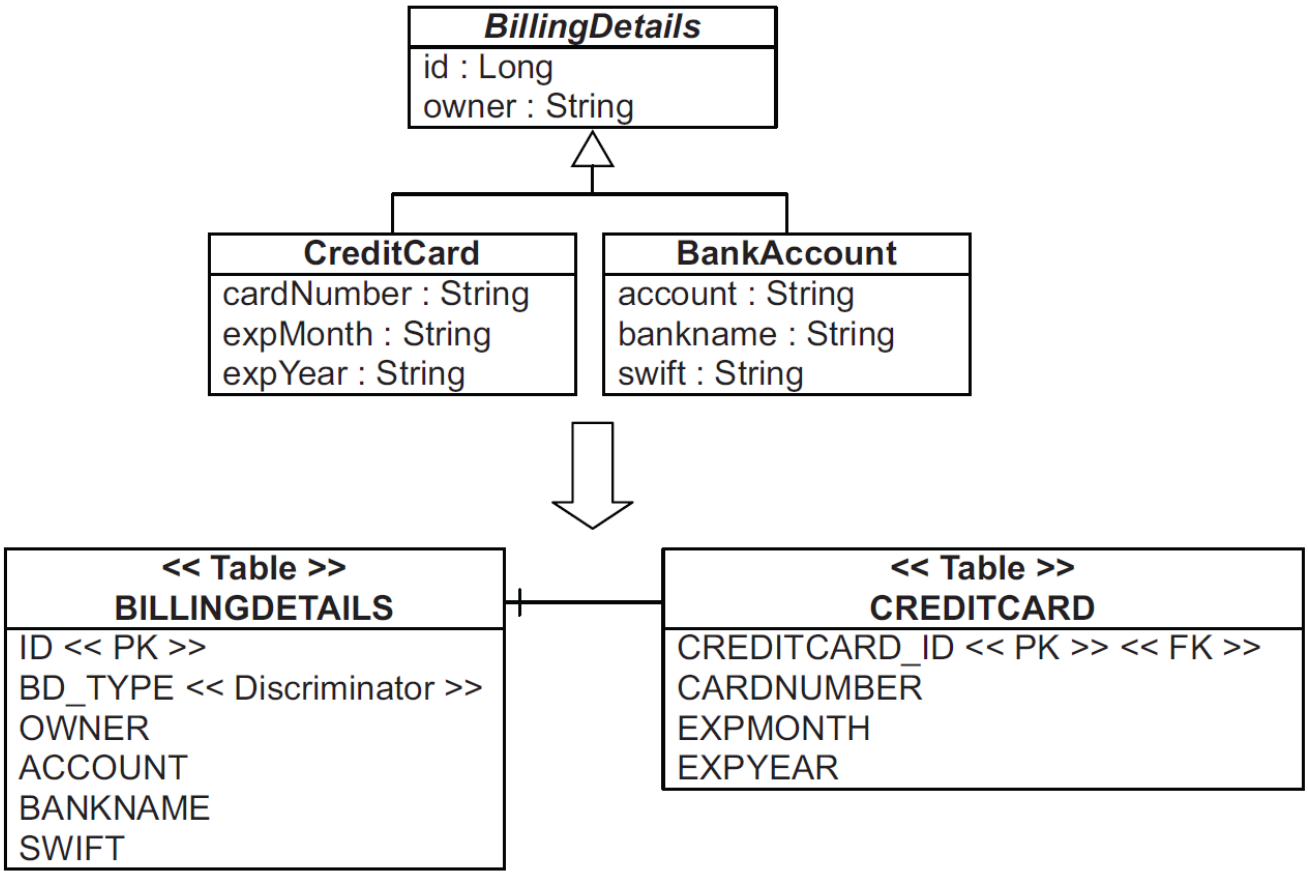
```
billingDetailsRepository.findAll();
creditCardRepository.findAll();
```

Сгенерируют:

```
select
    BD.ID, BD.OWNER,
    CC.EXPMONTH, CC.EXPYEAR, CC.CARDNUMBER,
    BA.ACCOUNT, BA.BANKNAME, BA.SWIFT,
    case
        when CC.CREDITCARD_ID is not null then 1
        when BA.ID is not null then 2
        when BD.ID is not null then 0
    end
from
    BILLINGDETAILS BD
    left outer join CREDITCARD CC on BD.ID=CC.CREDITCARD_ID
    left outer join BANKACCOUNT BA on BD.ID=BA.ID

select
    CREDITCARD_ID, OWNER, EXPMONTH, EXPYEAR, CARDNUMBER
from
    CREDITCARD
    inner join BILLINGDETAILS on CREDITCARD_ID=ID
```

Соответственно.

Смешивание стратегий отображений



```
@Entity
@DiscriminatorValue("CC")
@SecondaryTable(
name = "CREDITCARD",
pkJoinColumns = @PrimaryKeyJoinColumn(name = "CREDITCARD_ID")
)
public class CreditCard extends BillingDetails {
    @NotNull
    @Column(table = "CREDITCARD", nullable = false)
    private String cardNumber;
    @Column(table = "CREDITCARD", nullable = false)
    private String expMonth;
    @Column(table = "CREDITCARD", nullable = false)
    private String expYear;
    // . . .
}
```

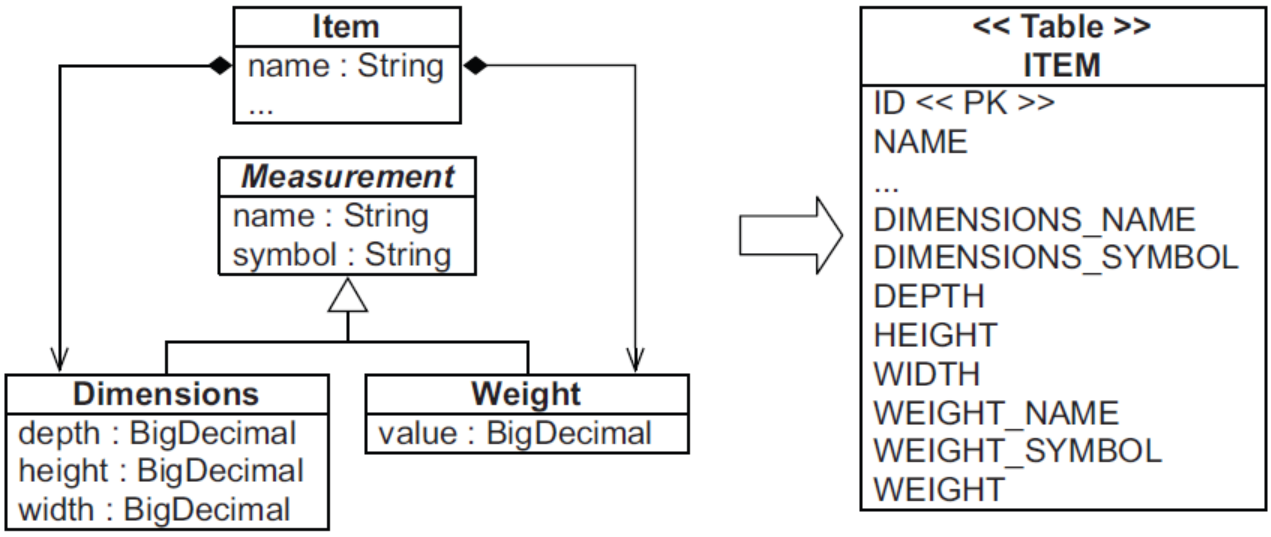
```
select
    ID, OWNER, ACCOUNT, BANKNAME, SWIFT,
    EXPMONTH, EXPYEAR, CARDNUMBER,
    BD_TYPE
from
    BILLINGDETAILS
    left outer join CREDITCARD on ID=CREDITCARD_ID
```

Наследование встроенных классов

```
@MappedSuperclass
public abstract class Measurement {
    @NotNull
    private String name;
    @NotNull
    private String symbol;
    // . . .
}
```

```
@Embeddable
@AttributeOverride(name = "name",
column = @Column(name = "DIMENSIONS_NAME"))
@AttributeOverride(name = "symbol",column = @Column(name = "DIMENSIONS_SYMBOL"))
public class Dimensions extends Measurement {
    @NotNull
    private BigDecimal depth;
    @NotNull
    private BigDecimal height;
```

```
    @NotNull
    private BigDecimal width;
    // . . .
}

@Embeddable
@AttributeOverride(name = "name",
column = @Column(name = "WEIGHT_NAME"))
@AttributeOverride(name = "symbol",
column = @Column(name = "WEIGHT_SYMBOL"))
public class Weight extends Measurement {
    @NotNull
    @Column(name = "WEIGHT")
    private BigDecimal value;
    // . . .
}

@Entity
public class Item {
    private Dimensions dimensions;
    private Weight weight;
    // . . .
}
```



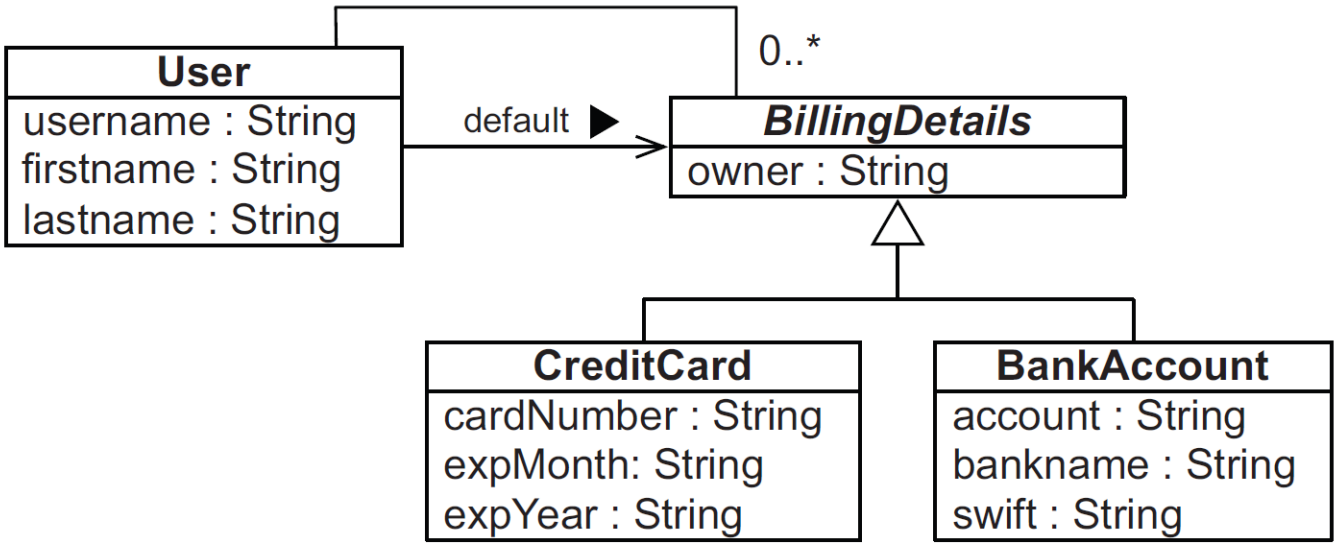
- Если вам не нужны полиморфные ассоциации или запросы, склоняйтесь к стратегии «таблица - конкретный класс» - другими словами, если вы никогда или редко делаете `select bd from BillingDetails bd`, и у вас нет класса, который бы ассоциировался с BillingDetails. Явное отображение на основе UNION с InheritanceType.TABLE_PER_CLASS должно быть предпочтительнее, поскольку (оптимизированные) полиморфные запросы и ассоциации будет возможно реализовать.

- Если вам требуются полиморфные ассоциации (ассоциации с суперклассом, а значит, и со всеми классами в иерархии с динамическим разрешением конкретного класса во время выполнения) или запросы, а подклассы объявляют относительно мало свойств (особенно если основное

различие между подклассами заключается в их поведении), склоняйтесь к InheritanceType.SINGLE_TABLE. Этот подход может быть выбран, если он предполагает установку малого количества столбцов как nullable.

- Если вам требуются полиморфные ассоциации или запросы, а подклассы объявляют много (неопциональных) свойств (подклассы отличаются в основном данными, которые они хранят), склоняйтесь к InheritanceType.JOINED. В качестве альтернативы, в зависимости от ширины и глубины иерархии наследования и возможных затрат на соединения по сравнению с союзами, используйте InheritanceType.TABLE_PER_CLASS. Для принятия такого решения может потребоваться оценка планов выполнения SQL с реальными данными.

Полиморфные ассоциации



Полиморфные многие-к-одному ассоциации

```
@Entity
@Table(name = "USERS")
public class User {
    @ManyToOne
    private BillingDetails defaultBilling;
    // . . .
}
```

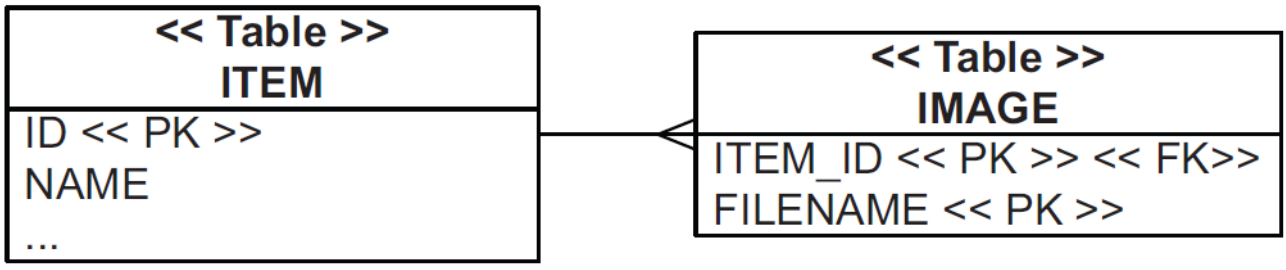
```
CreditCard creditCard = new CreditCard(
    "John Smith", "123456789", "10", "2030"
);

User john = new User("John Smith");
john.setDefaultBilling(creditCard);
creditCardRepository.save(creditCard);
userRepository.save(john);
```

```
...

List<User> users = userRepository.findAll();
users.get(0).getDefaultBilling().pay(123);
```

Полиморфные коллекции

```
@Entity
@Table(name = "USERS")
public class User {
    @OneToMany(mappedBy = "user")
    private Set<BillingDetails> billingDetails = new HashSet<>();
    // . . .
}

@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class BillingDetails {
    @ManyToOne
    private User user;
    // . . .
}
```

Отображение множеств




```
@Entity
public class Item {
    // . . .
    @ElementCollection
    @CollectionTable(
    name = "IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID"))
    @Column(name = "FILENAME")
    private Set<String> images = new HashSet<>();
```

```
public interface ItemRepository extends JpaRepository<Item, Long> {
    @Query("select i from Item i inner join fetch i.images where i.id = :id")
    Item findItemWithImages(@Param("id") Long id);

    @Query(value = "SELECT FILENAME FROM IMAGE WHERE ITEM_ID = ?1",
    nativeQuery = true)
    Set<String> findImagesNative(Long id);
}

...

@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {SpringDataConfiguration.class})
public class MappingCollectionsSpringDataJPATest {
    @Autowired
    private ItemRepository itemRepository;

    @Test
    void storeLoadEntities() {
        Item item = new Item("Foo");

        item.addImage("background.jpg");
        item.addImage("foreground.jpg");
        item.addImage("landscape.jpg");
        item.addImage("portrait.jpg");

        itemRepository.save(item);

        Item item2 = itemRepository.findItemWithImages(item.getId());
        List<Item> items2 = itemRepository.findAll();
        Set<String> images = itemRepository.findImagesNative(item.getId());

        assertAll(
        () -> assertEquals(4, item2.getImages().size()),
        () -> assertEquals(1, items2.size()),
        () -> assertEquals(4, images.size())
        );
    }
}
```

Отображение корзины с идентификаторами

Корзина — это неупорядоченная коллекция, допускающая дублирование элементов.

```
@Entity
public class Item {
    // . . .
    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
```

```
    @GenericGenerator(name = "sequence_gen", strategy = "sequence")
    @org.hibernate.annotations.CollectionId(
    columns = @Column(name = "IMAGE_ID"),
    type = @org.hibernate.annotations.Type(type = "long"),
    generator = "sequence_gen")
    private Collection<String> images = new ArrayList<>();
```

ITEM

ID	NAME
1	Foo

IMAGE

IMAGE_ID	ITEM_ID	FILENAME
1	1	landscape.jpg
2	1	foreground.jpg
3	1	background.jpg
4	1	portrait.jpg

Отображение списков

```
@Entity
public class Item {
    // . . .
    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @OrderColumn // Enables persistent order, Defaults to IMAGES_ORDER
    @Column(name = "FILENAME")
    private List<String> images = new ArrayList<>();
```

ITEM

ID	NAME
1	Foo
2	Bar

IMAGE

ITEM_ID	IMAGES_ORDER	FILENAME
1	0	landscape.jpg
1	1	foreground.jpg
1	2	background.jpg
1	3	background.jpg
2	0	portrait.jpg
2	1	foreground.jpg

Отображение словарей

```
@Entity
public class Item {
    // . . .
    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")
    @Column(name = "IMAGENAME")
    private Map<String, String> images = new HashMap<>();
```

ITEM

ID	NAME
1	Foo
2	Bar

IMAGE

ITEM_ID	FILENAME	IMAGENAME
1	landscape.jpg	Landscape
1	foreground.jpg	Foreground
1	background.jpg	Background
1	portrait.jpg	Portrait
2	landscape.jpg	Landscape
2	foreground.jpg	Foreground

Упорядоченные коллекции

```
@Entity
public class Item {
    // . . .
    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")
    @Column(name = "IMAGENAME")
    @org.hibernate.annotations.SortComparator(ReverseStringComparator.class)
    private SortedMap<String, String> images = new TreeMap<>();
```

```
@Entity
public class Item {
    // . . .
    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    @org.hibernate.annotations.SortNatural
    private SortedSet<String> images = new TreeSet< >();
```

```
@Entity
public class Item {
    // . . .
    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    // @javax.persistence.OrderBy // One possible order: "FILENAME asc"
    @org.hibernate.annotations.OrderBy(clause = "FILENAME desc")
    private Set<String> images = new LinkedHashSet<>();
```

```
@Entity
public class Item {
    // . . .
    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @Column(name = "FILENAME")
    @GenericGenerator(name = "sequence_gen", strategy = "sequence")
    @org.hibernate.annotations.CollectionId(
    columns = @Column(name = "IMAGE_ID"),
    type = @org.hibernate.annotations.Type(type = "long"),
    generator = "sequence_gen")
    @org.hibernate.annotations.OrderBy(clause = "FILENAME desc")
    private Collection<String> images = new ArrayList<>();
```

```
@Entity
public class Item {
    // . . .
    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @MapKeyColumn(name = "FILENAME")
    @Column(name = "IMAGENAME")
    @org.hibernate.annotations.OrderBy(clause = "FILENAME desc")
    private Map<String, String> images = new LinkedHashMap<>();
```

Отображение встраиваемых коллекций

```
@Embeddable
public class Image {
    @Column(nullable = false)
    private String filename;
    private int width;
    private int height;

    @org.hibernate.annotations.Parent
    private Item item;
```

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Image image = (Image) o;
        return width == image.width &&
            height == image.height &&
            filename.equals(image.filename) &&
            item.equals(image.item);
    }

    @Override
    public int hashCode() {
        return Objects.hash(filename, width, height, item);
    }
    // . . .
    }
}


...

@Entity
public class Item {
    // . . .
    @ElementCollection
    @CollectionTable(name = "IMAGE")
    @AttributeOverride(
        name = "filename",
        column = @Column(name = "FNAME", nullable = false)
    )
    private Set<Image> images = new HashSet<>();

...

@Query(value = "SELECT FNAME FROM IMAGE WHERE ITEM_ID = ?1", nativeQuery = true)
Set<String> findImagesNative(Long id);
```

ITEM

ID	NAME
1	Foo
2	Bar

IMAGE

ITEM_ID	FNAME	WIDTH	HEIGHT
1	landscape.jpg	640	480
1	foreground.jpg	800	600
1	background.jpg	1024	768
1	portrait.jpg	480	640
2	landscape.jpg	640	480
2	foreground.jpg	800	600