**ORM — Object-Relational Mapping или в переводе на русский объектно-реляционное отображение.** Это технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования. Если упростить, то ORM это связь Java объектов и записей в БД.

JPA (Java Persistence API) – это спецификация, которая означает набор рекомендаций, которым нужно следовать для представления Java-объектов в базах данных. JPA предоставляет набор концепций в виде интерфейсов и аннотаций для конфигурирования Java-объектов.

Инструменты ORM обеспечивают реализацию JPA: Hibernate, EclipseLink.

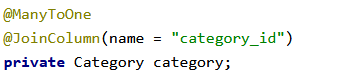
Зачем нужен JPA, ведь мы можем использовать инструменты ORM без спецификации JPA? Одна из причин – абстрагирование от конкретной реализации. JPA абстрагирует общие паттерны и определяет спецификации, чтобы поставщики ORM использовали их для создания конкретных решений.

Spring Data JPA не является провайдером JPA, это библиотека/фреймворк, которая добавляет дополнительный уровень абстракции поверх нашей линейки провайдеров JPA - Hibernate.

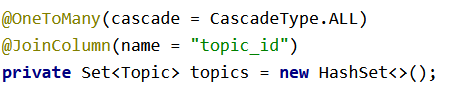
Отношения между Java-объектами устанавливаются путем добавления реляционных аннотаций

* One-to-one
* Many-to-many
* One-to-many
* Many-to-one

**@ManyToOne –** связь многие к одному. Когда есть главная сущность, и от нее зависят несколько других. Указывается над полем главной сущности в зависимом классе.

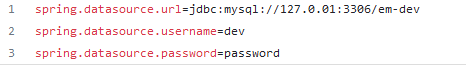


**@OneToMany(cascade = CascadeType.ALL) –** указывается над коллекцией зависимых сущностей. **cascade** определяет, что делать при удалении главной сущности.



**@JoinColumn(name=”collumn”)** – указывает по какому столбцу в бд устанавливается связь.

Чтобы подключиться к базе данных в файле свойств нужно указать url, username, password:



Так же может понадобиться JDBC-драйвер для подключения к БД, его нужно добавить в зависимости.

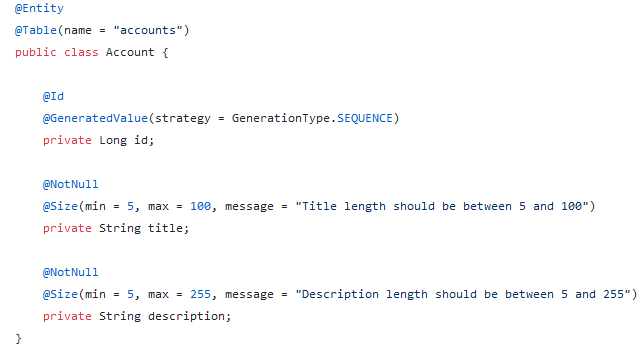
Основным элементом в JPA является класс **модели** или **сущности**. Этот класс сопоставляется с таблицей в базе данных.

Для маппинга используются следующие аннотации:

* @Entity: пометить класс как сущность БД.
* @Table: таблица в базе данных, на которую будет отображаться сущность. По умолчанию это имя класса. Можно указать в какой схеме лежит бд , schema="test".
* @Id: указывает первичный ключ сущности.
* @GenerateValue: стратегии геннерации id.

Стратегии генерации ID:

* TABLE – для генерации уникального значения используется отдельная таблица, которая эмулирует последовательность. Когда требуется новое значение, JPA провайдер блокирует строку таблицы, обновляет хранящееся там значение, и возвращает обратно в приложение. Эта стратегия наихудшая по производительности.
* SEQUENCE – используется последовательность, специальный объект БД для генерации уникальных значений.
* IDENTITY – используется встроенный в БД тип данных столбца – identity, для генерации значения первичного ключа.
* AUTO (по умолчанию) – JPA провайдер решает, как генерировать уникальные ID для сущности. Hibernate, например, сначала попробует использовать SEQUENCE. Если бд не поддерживает последовательности, то будет использоваться стратегия TABLE или IDENTITY.

   
**@Column –** аннотация, указывающая имя колонки в таблице. По умолчанию это имя поля.

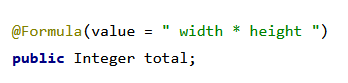
Параметры @Column:

* **name (**value**) –** имя колонки таблицы.
* **unique –** все значения поля должны быть уникальны.
* **nullable –** поле может принимать значения null.
* **length –** максимальная длина (для строк)

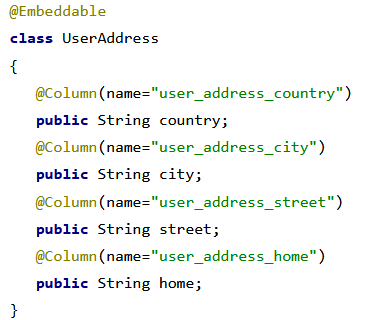
Если в БД нужно хранить **enum,** можно хранить его двумя способами:

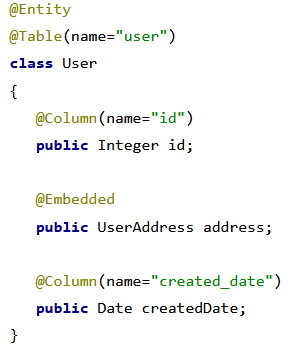
* В виде числа. Для этого нужно использовать над полем аннотацию  
  
* В виде строки. Использовать аннотацию  
  

**@Transient –** указывает что поле недолжно сериализоваться. Hibernate также будет игнорировать это поле.

**@Formula –** указывает, что поле объекта вычисляется на основе других полей.  


**@Embedded –** позволяет рассматривать поля дочернего объекта как поля самого Entity-класса. Дочерний класс при этом помечается аннотацией **@Embeddable**.





**@CreationTimestamp –** аннотация, указывающая что в поле хранится время создания объекта в базе.

**@UpdateTimestamp –** поле хранит время обновления сущности. Hibernate сам его обновляет.

Важно! Если ты решишь добавить в свой класс User два поля UserAddress, то использовать @Embedded уже не получится: у тебя будет дубликат полей и тебе нужно будет как-то их разделить. Делается это c помощью аннотации **@AttributeOverrides**.

Если мы хотим видеть какие sql запросы генерирует hibernate, то можно указать в конфигурации следующее:



параметр **hbm2ddl.auto** может иметь следующие значения:

|  |  |
| --- | --- |
| **validate** | Валидация: Hibernate проверит, совпадают ли имена и типа колонок и полей в базе и в аннотациях. Это самый частый режим. |
| **update** | Апдейт: Hibernate обновит таблицы в базе, если они или их колонки отличаются от ожидаемых. |
| **create** | Пересоздание: Hibernate удалит все таблицы в базе и создаст их заново на основе данных из аннотаций. |
| **create-drop** | Создание-удаление. В начале работы Hibernate создаст все таблицы, в конце работы – удалит их за собой. |
| **none** | Hibernate вообще ничего не будет делать. Если где-то база не совпадает с ожиданием, то будут сыпаться ошибки во время выполнения запросов. |

**@Repository –** указывает, что класс является репозиторием.

Когда мы расширяем JPA репозиторий, Spring предоставляет нам реализацию базовых методов CRUD. При этом по умолчанию выборки будут постраничные и отсортированные по первичному ключу.

В Hibernate сущности хранятся в PersistenceContext. Управление сущностями происходит через EntityManager. Чтобы сохранить сущность, мы вызываем у EntityManager метод persist. Эта сущность затем попадает в контекст.

Контекст – некоторая область хранения. Его иногда называют “кэшем первого уровня”. Чтобы данные из контекста попали в бд, используется команда **flush**. Эта команда должна выполняться в рамках транзакции.



При выборке данных из бд, менеджер сущностей сначала ищет сущность в контексте (своего рода кэш), и только если не находит – идет искать в бд.

При использовании .properties SessionFactory можно получить следующим образом:



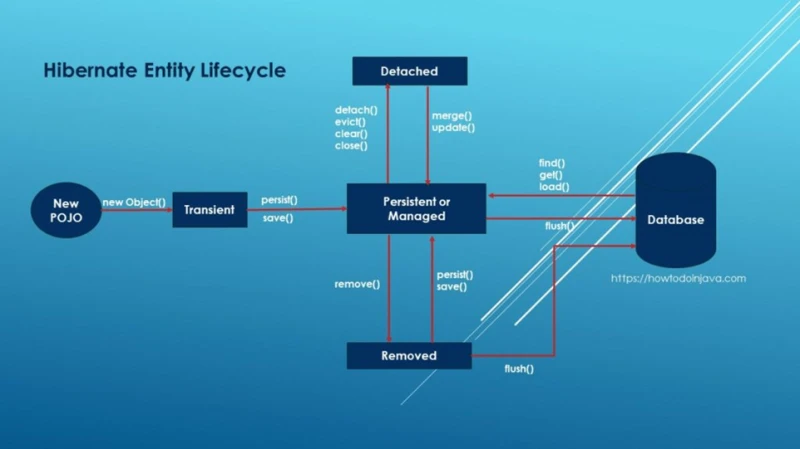
При конфигурации с помощью **hibernate.cfg.xml:**



**Жизненный цикл сущностей**

У сущности может быть 4 состояния:

* **Transient**
* **Persistent** (or Managed)
* **Detached**
* **Removed**



**Transient –** каждый Entity объект, который создан явно с помощью Java-кода, а не загруженный из БД с помощью Hibernate. Это значит, что Hibernate понятия не имеет об этом объекте, и никакие действия с ним не влияют на Hibernate.

**Persistent or Managed** – объекты, отслеживаемые Hibernate. Получить их можно двумя способами:

* Загрузить объект из Hibernate
* Сохранить объект в Hibernate

Такому объекту обычно соответствует какая-то запись в базе данных, у него есть id и т.п.

**Detached –** объект отсоединен от сессии. Когда-то объект был присоединен к сессии, но затем сессия закрылась или транзакция завершилась, и hibernate больше не следит за этим объектом.

Вызвав метод **evict** объект будет явно отсоединен от сессии.

**Removed** – состояние удаленного объекта. Когда мы удаляем какой-то объект из базы, то Java-объект сразу никуда не исчезнет.

**Session –** основной объект для работы с базой данных. В нем есть все методы EntityManger. Но также есть и свои, которые достались ему от более ранних версий Hibernate, когда еще не было JPA.

**Методы session:**

* session.persist(obj) – сохранить объект в бд. Если такого объекта еще нет, то он будет добавлен в базу через вызов SQL-метода INSERT.

Если объект уже есть в базе, то ничего не произойдет. Если попытаться сохранить объект со статусом Detached, то выбросится исключение PersistenceException.

* session.save(obj) – метод из старых версий hibernate. Возвращает ID, который был присвоен сохраняемому объекту базой данных. Согласно спецификации Hibernate id может быть любой сериализуемый объект, что может быть целиком положено в одну колонку таблицы в базе.

Метод save() возвращает Serialized, поэтому его нужно приводить к нужному типу. Если в save() передать detached объект, то он будет рассмотрен как новый и будет добавлена еще одна запись в бд.

Если объект в статусе Persist, то нет необходимости каждый раз сохранять объект после изменений. Hibernate сам записывает в базу все изменения отслеживаемого объекта.

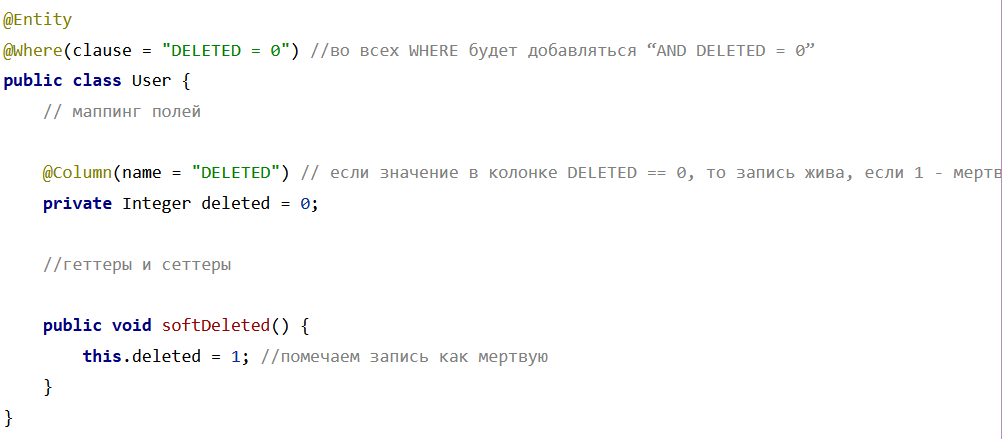
* Session.merge(obj) – обновляет информацию в базе на основе переданного объекта. Возвращает обновленный объект, который будет иметь состояния persist и присоединен к объекту session.

При этом объект, переданный в merge() при этом не меняется.

Если передаваемый в merge() объект имеет статус Transient (и у него нет ID), то для него создается новая строка в бд.

* session.update(obj) – метод из старых версий hibernate. Этот метод ничего не возвращает и не меняет существующий объект. Если вызвать этот метод для нового объекта, то просто кинется исключение.
* Session.saveOrUpdate(obj) – старая альтернатива метода persist().Выполняет обновление объекта, а если его нет в бд, то создает его. В отличие от update() может менять переданный ему объект, например, установить ему ID.
* session.get(EntityClass.class, Object primaryKey) – возвращает объект из БД, указанного класса по id. Если запись с таким ID не найдена, возвращает null.
* session.load(EntityClass.class, Object primaryKey) – загружает объект из бд. Возвращает proxy. Если объекта нет в бд, Hibernate создает proxy-объект с переданным id и возвращает его. Вся работа с базой данных будет происходить при вызове методов. При попытке доступа к свойствам и произойдет первое обращение к базе. Если в него передать невалидный id, метод вернет null.
* session.find(User.class, id) – метод стандарта JPA. Если объект не найден, возвращает null.
* session.refresh(obj) – обновляет существующий объект на основе данных из БД. Такое поведение может быть необходимо, если при записи объекта в бд вызываются различные хранимые процедуры, которые корректируют записанные данные.
* session.remove(obj) – удалить объект из бд. Реальная операция в базе будет выполнена после вызова метода flush() или закрытия транзакции.

Если нужно “мягкое удаление” (не удалять данные, а просто пометить их как удаленные) то можно завести поле, которое будет указывать удален объект или нет. Hibernate имеет аннотацию **@Where**, текст которой будет добавляться в каждый запрос.



**Criteria API**

https://javarush.com/groups/posts/2259-jpa--znakomstvo-s-tekhnologiey