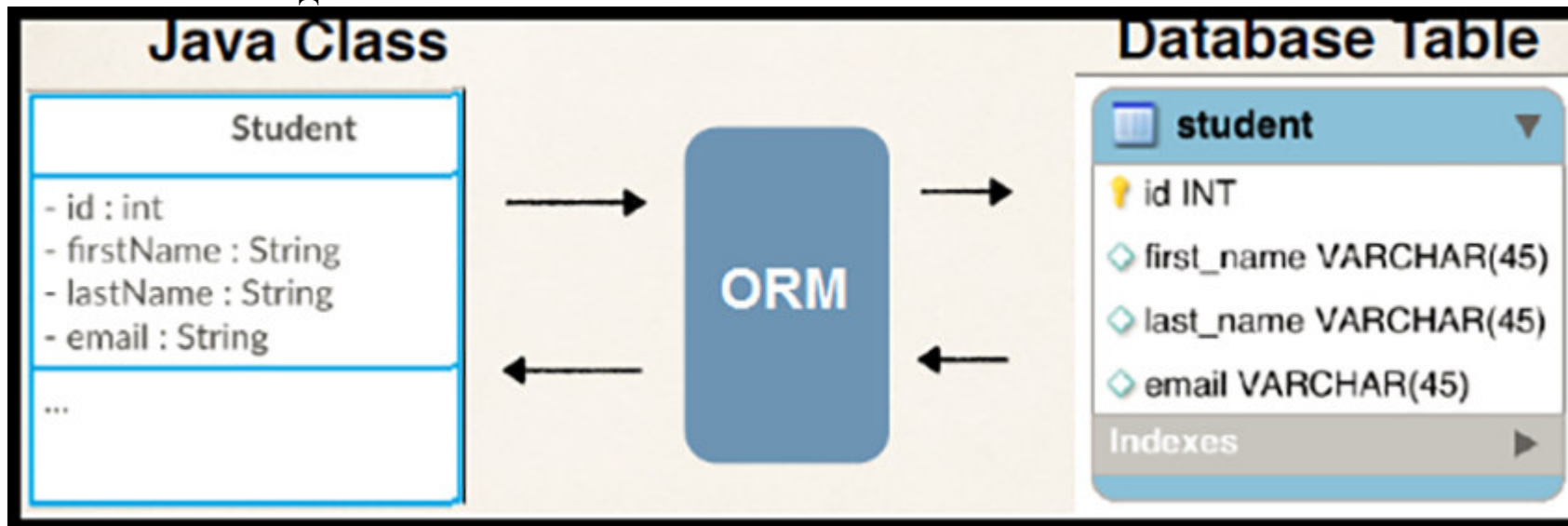


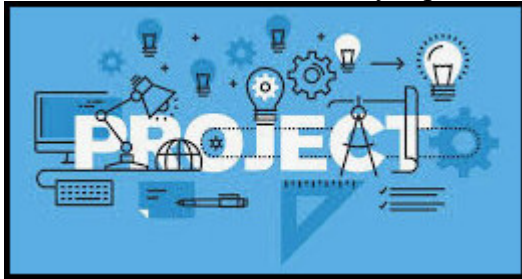
Вступление

Как мы знаем, одна из основных задач программ — хранение и обработка данных. В старые добрые времена люди хранили данные просто в файлах. Но как только нужен одновременный доступ на чтение и редактирование, когда появляется нагрузка (т.е. одновременно поступает несколько обращений), хранение данных просто в файлах становится проблемой. Подробнее о том, какие проблемы решают БД и каким образом, советую прочитать в статье "[Как устроены базы данных](#)". Значит, наши данные мы решаем хранить в базе данных. С давних пор Java умеет работать с базами данных при помощи JDBC API (The Java Database Connectivity). Подробнее про JDBC можно прочитать тут: "[JDBC или с чего всё начинается](#)". Но время шло и разработчики каждый раз сталкивались с необходимостью писать однотипный и ненужный "обслуживающий" код (так называемый Boilerplate code) для тривиальных операций по сохранению Java объектов в БД и наоборот, созданию Java объектов по данным из БД. И тогда для решения этих проблем на свет появилось такое понятие, как ORM. **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:



ORM — это по сути концепция о том, что Java объект можно представить как данные в БД (и наоборот). Она нашла воплощение в виде спецификации JPA — Java Persistence API. Спецификация — это уже описание Java API, которое выражает эту концепцию. Спецификация рассказывает, какими средствами мы должны быть обеспечены (т.е. через какие интерфейсы мы сможем работать), чтобы работать по концепции ORM. И как использовать эти средства. Реализацию средств спецификация не описывает. Это даёт возможность использовать для одной спецификации

разные реализации. Можно упростить и сказать, что спецификация — это описание API. Текст спецификации JPA можно найти на сайте Oracle: "[JSR 338: Java™ Persistence API](#)". Следовательно, чтобы использовать JPA нам требуется некоторая реализация, при помощи которой мы будем пользоваться технологией. Реализации JPA ещё называют JPA Provider. Одной из самых заметных реализаций JPA является [Hibernate](#). Поэтому, предлагаю её и рассмотреть.

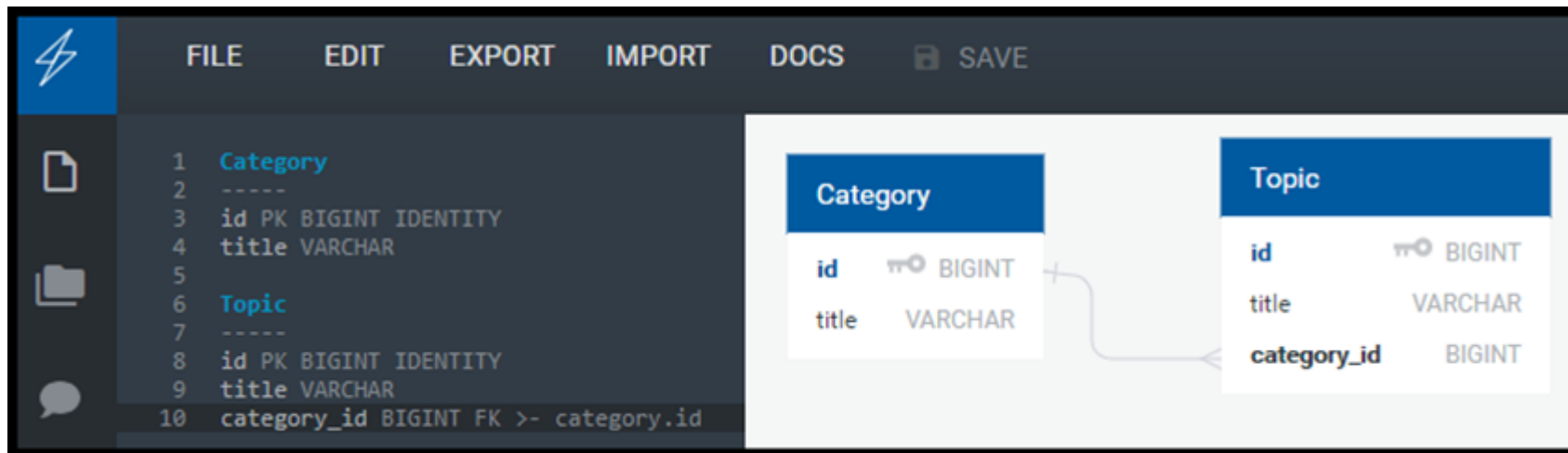


Создание проекта

Так как JPA — это про Java, то нам понадобится Java проект. Мы могли бы сами вручную создать структуру каталогов, сами добавить нужные библиотеки. Но куда удобнее и правильнее использовать системы автоматизации сборки проектов (т.е. по сути это просто программа, которая за нас будет управлять сборкой проектов. Создавать каталоги, подкладывать в classpath нужные библиотеки и т.д.). Одной из такой систем является Gradle. Подробнее про Gradle можно прочитать здесь: "[Краткое знакомство с Gradle](#)". Как мы знаем, функциональность Gradle (т.е. действия, которые он может сделать) реализованы при помощи различных Gradle Plugins. Воспользуемся Gradle и плагином "[Gradle Build Init Plugin'om](#)". Выполним команду:

```
gradle init --type java-application
```

Gradle за нас сделает нужную структуру каталогов, создаст базовое декларативное описание проекта в билд скрипте `build.gradle`. Итак, у нас появилось приложение. Нам надо подумать, что мы хотим описывать или моделировать нашим приложением. Давайте воспользуемся каким-нибудь средством моделирования, например: app.quickdatabasediagrams.com



Тут стоит

сказать, что то что мы описали нашу "доменную модель". Домен — это некоторая "предметная область". Вообще, домен — это "владение" на латыни. В средние века так назывались области, которыми владели короли или феодалы. А во французском языке это стало словом "domaine", которое переводится просто как "область". Таким образом мы описали нашу "доменную модель" = "предметную модель". Каждый элемент этой модели — это некоторая "сущность", что-то из реальной жизни. В нашем случае это сущности: Категория (Category), Тема (Topic). Создадим для сущностей отдельный пакет, например с именем model. И добавим туда Java классы, описывающие сущности. В Java коде такие сущности представляют из себя обычный [POJO](#), который может выглядеть так:

```
public class Category {
    private Long id;
    private String title;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

Скопируем содержимое класса и сделаем по аналогии класс Topic. Отличаться он будет лишь тем, что он знает про категорию, к которой относится. Поэтому, добавим в класс Topic поле категории и методы работы с ней:

```
private Category category;

public Category getCategory() {
```

```
        return category;
    }

    public void setCategory(Category category) {
        this.category = category;
    }
}
```

Теперь у нас есть Java приложение, имеющее свою доменную модель. Пора теперь приступить к подключению к проекту JPA.



Добавление JPA

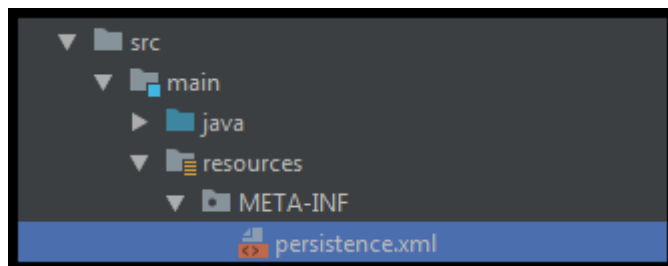
Итак, как мы помним, JPA — это про то, что мы будем сохранять что-то в БД. Следовательно, нам нужна база данных. Чтобы использовать подключение к БД в своём проекте нам нужно добавить в зависимости библиотеку, для подключения к БД. Как мы помним, мы использовали Gradle, который создал нам билд скрипт `build.gradle`. В нём то мы и опишем зависимости, которые нужны нашему проекту. Зависимости — это те библиотеки, без которых не может работать наш код. Начнём с описания зависимости от подключения к БД. Делаем это так же, как бы делали, работая просто с JDBC:

```
dependencies {
    implementation 'com.h2database:h2:1.4.199'
```

Теперь у нас есть БД. Мы можем теперь добавить в наше приложение уровень или слой (layer), отвечающий за отображение наших Java объектов в понятия базы данных (с Java языка на язык SQL). Как мы помним, мы собираемся использовать для этого реализацию спецификации JPA под названием Hibernate:

```
dependencies {
    implementation 'com.h2database:h2:1.4.199'
    implementation 'org.hibernate:hibernate-core:5.4.2.Final'
```

Теперь нам надо сконфигурировать JPA. Если мы прочитаем спецификацию и раздел "8.1 Persistence Unit", то мы узнаем, что Persistence Unit — это некоторая некоторое объединение конфигураций, метаданных и сущностей. И чтобы JPA заработал, нужно описать хотя бы один Persistence Unit в конфигурационном файле, который имеет название `persistence.xml`. Его расположение описано в главе спецификации "8.2 Persistence Unit Packaging". Согласно этому разделу, если у нас Java SE окружение, то мы должны положить его в корень каталога META-INF.



Содержание скопируем из примера, приведённого в спецификации JPA в главе "8.2.1 persistence.xml file":

```
<persistence>
    <persistence-unit name="JavaRush">
        <description>Persistence Unit For test</description>
        <class>hibernate.model.Category</class>
        <class>hibernate.model.Topic</class>
    </persistence-unit>
</persistence>
```

Но этого мало. Надо рассказать, кто наш JPA Provider, т.е. тот, кто реализует спецификацию JPA:

```
<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
```

А теперь добавим настройки (properties). Часть из них (начинаются на javax.persistence) являются стандартными JPA конфигурациями и описаны в спецификации JPA в разделе "8.2.1.9 properties". Часть конфигураций являются провайдер-специфичными (в нашем случае, влияют на Hibernate как на Jpa Provider'a. Наш блок настроек будет выглядеть так:

```
<properties>
    <property name="javax.persistence.jdbc.driver" value="org.h2.Driver" />
    <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE" />
    <property name="javax.persistence.jdbc.user" value="sa" />
    <property name="javax.persistence.jdbc.password" value="" />
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.hbm2ddl.auto" value="create" />
</properties>
```

Теперь у нас есть JPA-совместимый конфиг persistence.xml, есть JPA провайдер Hibernate и есть база данных H2, а так же есть 2 класса, который являются нашей доменной моделью. Давайте заставим это всё наконец-то отработать. В каталоге /test/java наш Gradle любезно нам сгенерировал шаблон для Unit тестов и назвал его AppTest. Давайте используем его. Как гласит глава "7.1 Persistence Contexts" спецификации JPA, сущности в мире JPA живут в некотором пространстве, которое называется "Контекст персистенции" (или Контексте постоянства, Persistence Context). Но напрямую мы не работаем с Persistence Context. Для этого мы используем Entity Manager или "менеджер сущностей". Именно он знает про контекст и про то, какие там живут сущности. Мы же взаимодействуем с Entity Manager'ом. Тогда остаётся только понять, откуда нам достать этот Entity Manager? Согласно главе "7.2.2 Obtaining an Application-managed Entity

Manager" спецификации JPA мы должны использовать `EntityManagerFactory`. Поэтому, вооружимся спецификацией JPA и возьмём пример из главы "7.3.2 Obtaining an Entity Manager Factory in a Java SE Environment" и оформим его в виде простейшего Unit теста:

```
@Test
public void shouldStartHibernate() {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory( "JavaRush" );
    EntityManager entityManager = emf.createEntityManager();
}
```

Уже этот тест покажет ошибку "Unrecognized JPA persistence.xml XSD version". Причина — в `persistence.xml` нужно правильно указать используемую схему, как это сказано в спецификации JPA в разделе "8.3 persistence.xml Schema":

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
    version="2.2">
```

Кроме того, важен порядок элементов. Поэтому, `provider` должен быть указан до перечисления классов. После этого тест выполнится успешно. Непосредственное подключение JPA мы выполнили. Прежде чем мы будем двигаться дальше, подумаем про остальные тесты. Каждый наш тест будет требовать `EntityManager`. Давайте сделаем так, чтобы у каждого теста был свой `EntityManager` на начало выполнения. Кроме того, мы хотим чтобы БД каждый раз была новая. Благодаря тому, что мы используем `inmemory` вариант, достаточно закрывать `EntityManagerFactory`. Создание `Factory` — дорогая операция. Но для тестов — это оправдано. JUnit позволяет задать методы, которые будут выполняться перед (Before) и после (After) выполнением каждого теста:

```
public class AppTest {
    private EntityManager em;

    @Before
    public void init() {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory( "JavaRush" );
        em = emf.createEntityManager();
    }

    @After
    public void close() {
        em.getEntityManagerFactory().close();
        em.close();
    }
}
```

Теперь, перед выполнением любого теста будет создана новая `EntityManagerFactory`, что повлечёт за собой создание новой БД, т.к. `hibernate.hbm2ddl.auto` имеет значение `create`. А из новой фабрики получим новый `EntityManager`.

ENTITIES

Сущности (Entities)

Как мы помним, мы создали ранее классы, описывающие нашу доменную модель. Мы уже говорили, что это наши "сущности". Это и есть Entity, которыми мы будем управлять при помощи EntityManager. Напишем простой тест по сохранению сущности категории:

```
@Test
public void shouldPersistCategory() {
    Category cat = new Category();
    cat.setTitle("new category");
    // JUnit обеспечит тест свежим EntityManager'ом
    em.persist(cat);
}
```

Но сразу этот тест не заработает, т.к. мы получим различные ошибки, которые нам помогут понять, что такое сущности:

- Unknown entity: hibernate.model.Category
Почему же Hibernate не понимает, что Category это entity? Всё дело в том, что сущности должны быть описаны по стандарту JPA. Классы сущностей должны быть аннотированы аннотацией @Entity, как сказано в главе "2.1 The Entity Class" спецификации JPA.
- No identifier specified for entity: hibernate.model.Category
Для сущностей должен быть указан уникальный идентификатор, по которому можно отличить одну запись от другой. Согласно главе "2.4 Primary Keys and Entity Identity" спецификации JPA "Every entity must have a primary key", т.е. каждая сущность должна иметь "первичный ключ". Такой первичный ключ должен быть указан аннотацией @Id
- ids for this class must be manually assigned before calling save()
Идентификатор должен откуда-то появиться. Его можно указать вручную, а можно получить автоматически. Поэтому, как и указано в главах "11.2.3.3 GeneratedValue" и "11.1.20 GeneratedValue Annotation", мы можем указать аннотацию @GeneratedValue.

Таким образом, чтобы класс категории стал сущностью мы должны выполнить следующие изменения:

```
@Entity
public class Category {
    @Id
```

```
@GeneratedValue  
private Long id;
```

Кроме того, аннотация `@Id` указывает на то, какой использовать `Access Type`. Подробнее про тип доступа можно прочитать в спецификации JPA, в разделе "2.3 Access Type". Если очень кратко, то т.к. мы указали `@Id` над полем (field), то тип доступа будет по умолчанию `field-based`, а не `property-based`. Следовательно, провайдер JPA будет читать и сохранять значения напрямую из полей. Если бы мы поместили `@Id` над геттером, то использовался бы `property-based` доступ, т.е. через геттер и сеттер. При выполнении теста мы видим в том числе то, какие запросы отправляются в базу (благодаря опции `hibernate.show_sql`). Но при сохранении мы не видим никаких `insert`'ов. Получается, что мы на самом деле ничего не сохранили? JPA позволяет синхронизировать контекст персистенции и БД при помощи метода `flush`:

```
entityManager.flush();
```

Но если мы его сейчас выполним, то получим ошибку: *no transaction is in progress*. И тут наступает пора узнать про то, как JPA использует транзакции.



JPA Transactions

Как мы помним, в основе JPA лежит понятие контекст персистенции (Persistence Context). Это место, где живут сущности. А мы управляем сущностями через `EntityManager`. Когда мы выполняем команду `persist`, то мы помещаем сущность в контекст. Точнее, мы говорим `EntityManager`'у, что это нужно сделать. Но контекст этот — это просто некоторая область хранения. Его даже иногда называют "кэшем первого уровня". Но его нужно соединить с базой данных. Команда `flush`, которая ранее у нас упала с ошибкой, синхронизирует данные из контекста персистенции с БД. Но для этого требуется транспорт и этим транспортом является транзакция. Транзакции в JPA описаны в разделе спецификации "7.5 Controlling Transactions". Для использования транзакций в JPA есть специальный API:

```
entityManager.getTransaction().begin();  
entityManager.getTransaction().commit();
```

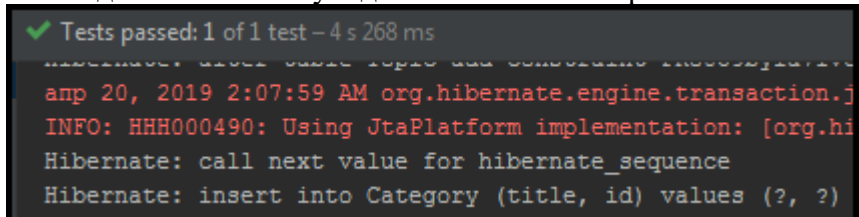
Необходимо добавить управление транзакциями в наш код, который выполняется до тестов и после:


```

@Before
public void init() {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory( "JavaRush" );
    em = emf.createEntityManager();
    em.getTransaction().begin();
}
@After
public void close() {
    if (em.getTransaction().isActive()) {
        em.getTransaction().commit();
    }
    em.getEntityManagerFactory().close();
    em.close();
}

```

После добавления мы увидим в логге insert выражение на языке SQL, которых ранее не было:



```

✓ Tests passed: 1 of 1 test - 4 s 268 ms
amp 20, 2019 2:07:59 AM org.hibernate.engine.transaction.j
INFO: HHH000490: Using JtaPlatform implementation: [org.hi
Hibernate: call next value for hibernate_sequence
Hibernate: insert into Category (title, id) values (?, ?)

```

Изменения, накопленные в EntityManager было при помощи транзакции закоммичены (подтверждены и сохранены) в БД. Давайте попробуем теперь найти нашу сущность. Создадим тест на поиск сущности по её ID:

```

@Test
public void shouldFindCategory() {
    Category cat = new Category();
    cat.setTitle("test");
    em.persist(cat);
    Category result = em.find(Category.class, 1L);
    assertNotNull(result);
}

```

В этом случае мы получим ранее сохранённую нами сущность, но в логге мы не увидим SELECT запросов. А всё по тому, что мы говорим: "Менеджер сущностей, найди пожалуйста мне сущность Категория с ID=1". А менеджер сущностей сначала смотрит у себя в контексте (использует его своего рода кэш), и только если не находит — идёт искать в БД. Стоит изменить ID на 2 (такого нет, мы сохранили только 1 экземпляр), как мы увидим, что SELECT запрос появляется. Потому что в контексте не найдено сущностей и EntityManager пытается найти сущность БД.. Существуют разные команды, которыми мы можем управлять состоянием сущности в контексте. Переход сущности из одного состояния в другое называется жизненным циклом сущности — lifecycle.

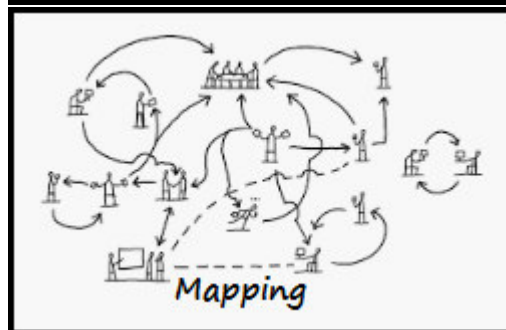
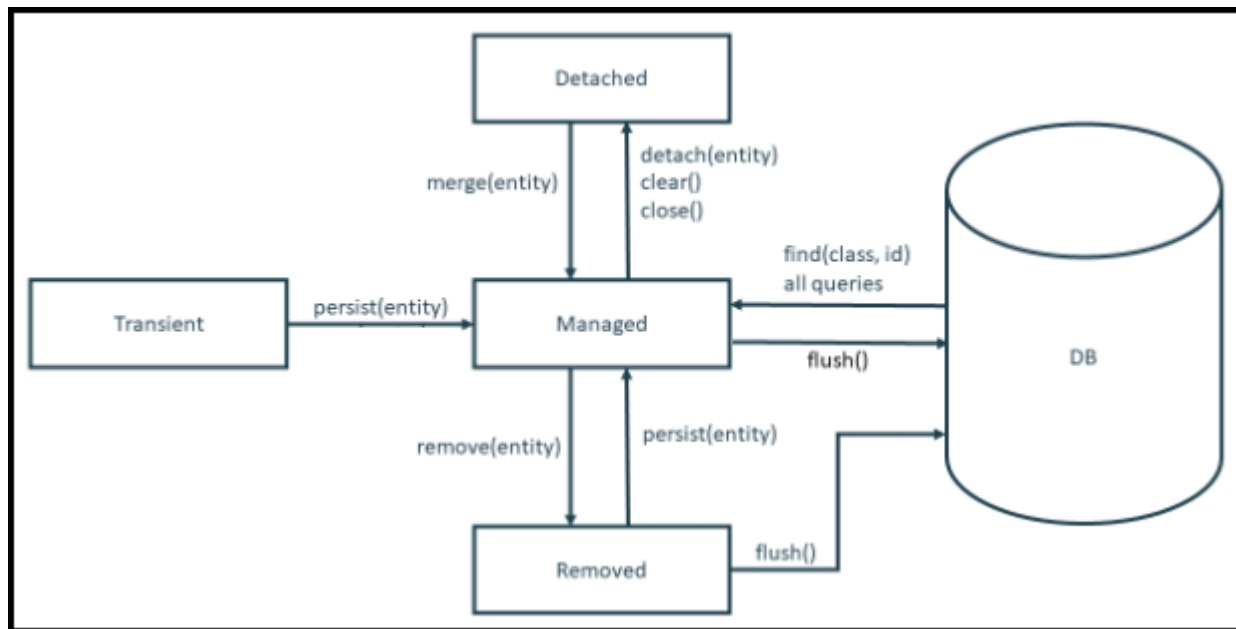


Entity Lifecycle

Жизненный цикл сущностей описан в спецификации JPA в главе "3.2 Entity Instance's Life Cycle". Т.к. сущности живут в контексте и ими управляет EntityManager, то говорят, что сущности управляемые, т.е. managed. Давайте посмотрим на этапы жизни сущности:

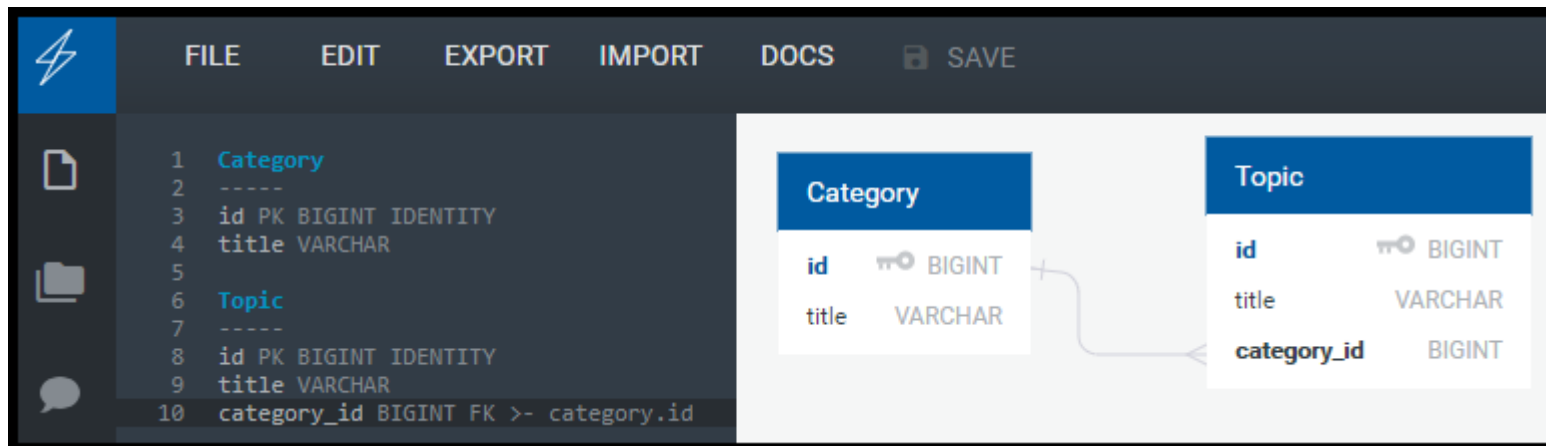
```
// 1. New или Transient (временный)
Category cat = new Category();
cat.setTitle("new category");
// 2. Managed или Persistent
entityManager.persist(cat);
// 3. Транзакция завершена, все сущности в контексте detached
entityManager.getTransaction().begin();
entityManager.getTransaction().commit();
// 4. Сущность изымаем из контекста, она становится detached
entityManager.detach(cat);
// 5. Сущность из detached можно снова сделать managed
Category managed = entityManager.merge(cat);
// 6. И можно сделать Removed. Интересно, что cat всё равно detached
entityManager.remove(managed);
```

И вот для закрепления схемка:

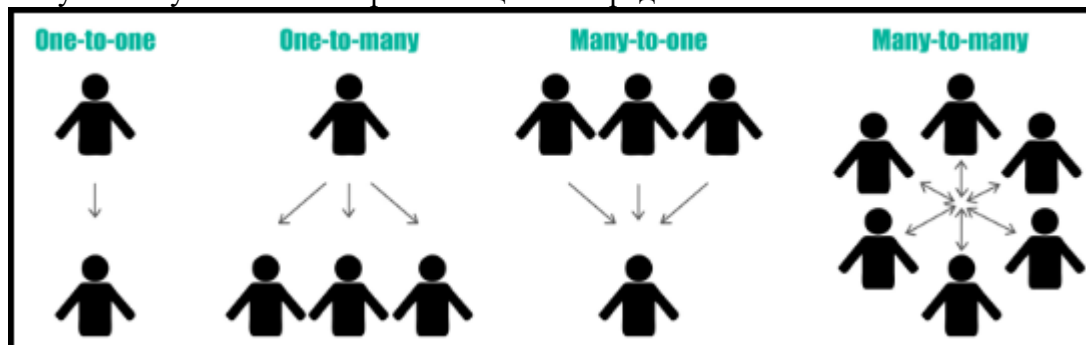


Mapping

В JPA мы можем описать отношения сущностей между друг другом. Вспомним, что мы уже разбирали отношения сущностей между друг другом, когда мы разбирались с нашей доменной моделью. Тогда мы использовали ресурс quickdatabasediagrams.com:



Установление связей между сущностями называется маппингом или ассоциированием (Association Mappings). Виды ассоциаций, которые могут быть установлены при помощи JPA представлены ниже:



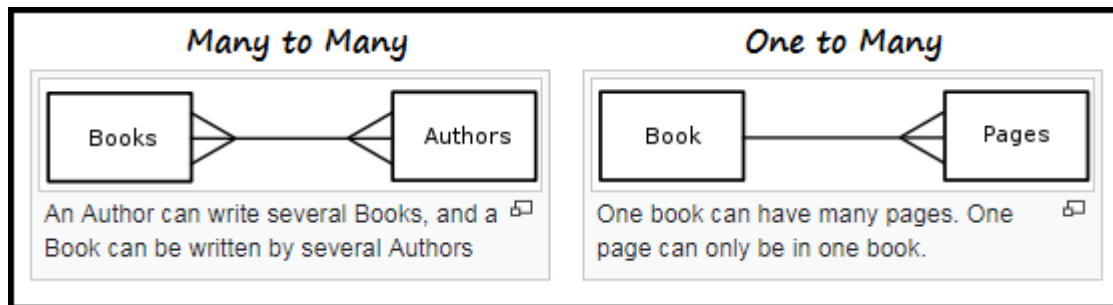
Давайте посмотрим на сущность `Topic`, которая описывает тему. Что мы можем сказать про отношение `Topic` к `Category`? Много `Topic` будут принадлежать одной категории. Следовательно, нам нужна ассоциация `ManyToOne`. Выразим эту связь на языке JPA:

```

@ManyToOne
@JoinColumn(name = "category_id")
private Category category;

```

Чтобы запомнить, какие аннотации ставить, можно запомнить, что последняя часть отвечает за поле, над которым указана аннотация. `ToOne` — конкретный экземпляр. `ToMany` — коллекции. Сейчас у нас связь односторонняя. Давайте сделаем из неё двустороннюю связь. Добавим в `Category` знание о всех `Topic`, которые входят в эту категорию. Оканчиваться должен на `ToMany`, потому что у нас список `Topic`. То есть отношение "Ко многим" темам. Остаётся вопрос — `OneToMany` или `ManyToMany`:



На эту же тему хороший ответ можно прочитать тут: ["Explain ORM oneToMany, manyToMany relation like I'm five"](#). Если категория имеет связь с ToMany топиков, то каждый из этих топиков может иметь только одну категорию, то будет One, а иначе Many. Таким образом, в Category список всех тем будет выглядеть следующий образом:

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "topic_id")
private Set<Topic> topics = new HashSet<>();
```

И не забудем в сущности Category описать геттер для получения списка всех тем:

```
public Set<Topic> getTopics() {
    return this.topics;
}
```

Двунаправленные отношения — очень сложный для автоматического отслеживания момент. Поэтому, JPA перекладывает эту обязанность на разработчика. Для нас это означает, что когда мы устанавливаем в сущности Topic связь с Category, мы должны обеспечить непротиворечивость данных самостоятельно. Делается это просто:

```
public void setCategory(Category category) {
    category.getTopics().add(this);
    this.category = category;
}
```

Напишем для проверки простой тест:

```
@Test
public void shouldPersistCategoryAndTopics() {
    Category cat = new Category();
    cat.setTitle("test");
    Topic topic = new Topic();
    topic.setTitle("topic");
    topic.setCategory(cat);
    em.persist(cat);
}
```

Маппинг — это целая отдельная тема. В рамках данного обзора следует понять, при помощи каких средств это достигается. Подробнее про маппинг можно прочитать тут:

- ["Ultimate Guide – Association Mappings with JPA and Hibernate"](#).
- ["JPA и связи между объектами"](#)
- ["Связанные сущности в Hibernate"](#)



JPQL

JPA вводит интересный инструмент — запросы на языке Java Persistence Query Language. Этот язык похож на SQL, но использует объектную модель Java, а не SQL таблицы. Рассмотрим пример:

```
@Test
public void shouldPerformQuery() {
    Category cat = new Category();
    cat.setTitle("query");
    em.persist(cat);
    Query query = em.createQuery("SELECT c from Category c WHERE c.title = 'query'");
    assertNotNull(query.getSingleResult());
}
```

Как мы видим, в запросе мы использовали указание на сущность `Category`, а не таблицу. А так же на поле этой сущности `title`. JPQL предоставляет множество полезных возможностей и претендует на отдельную статью. Подробнее можно ознакомиться в обзоре:

- ["Ultimate Guide to JPQL Queries with JPA and Hibernate"](#)



Criteria API

И напоследок хотелось бы затронуть Criteria API. JPA вводит инструмент динамического построения запросов. Пример использования Criteria API:

```
@Test
public void shouldFindWithCriteriaAPI() {
    Category cat = new Category();
    em.persist(cat);
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Category> query = cb.createQuery(Category.class);
    Root<Category> c = query.from(Category.class);
    query.select(c);
    List<Category> resultList = em.createQuery(query).getResultList();
    assertEquals(1, resultList.size());
}
```

Данный пример равносителен выполнению запроса "SELECT c FROM Category c". **Criteria API** — мощный инструмент. Подробнее про него можно прочитать здесь:

- [JPA Criteria API Queries](#)
- [JPA Criteria](#)
- [Динамические типобезопасные запросы в JPA 2.0](#)

Заключение

Как мы видим, JPA предоставляет огромное количество возможностей и инструментов. Каждый из них требует опыта и знаний. Даже в рамках обзора JPA вышло упомянуть не всё, не говоря уже о детальном погружении. Но надеюсь, после прочтения стало понятнее, что вообще такое ORM и JPA, как это работает и что с этим можно сделать.