Java Pro 7.13

TEL-RAN
by Starta Institute

Spring Data JPA

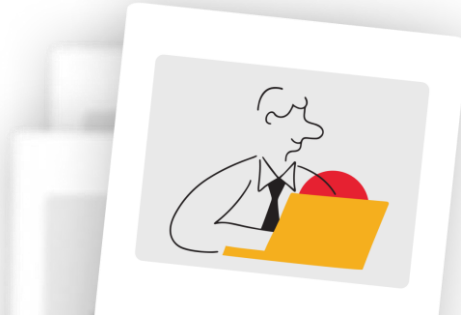
ПРЕПОДАВАТЕЛЬ

TEL-RAN
by Starta Institute

Юрий Костяной

Java/Kotlin backend-разработчик

- 3+ года опыта в коммерческой разработке
- 2+ года опыта в преподавании
- Проекты по интеграции сторонних платформ, CRM
- Проблемно-ориентированный подход в преподавании



ВАЖНО:

- Камера должна быть включена на протяжении всего занятия.
- Если у Вас возник вопрос в процессе занятия, пожалуйста, поднимите руку и дождитесь, пока преподаватель закончит мысль и спросит Вас, также можно задать вопрос в чате или когда преподаватель скажет, что начался блок вопросов.
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях.
- Вести себя уважительно и этично по отношению к остальным участникам занятия.
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя.

TEL-RAN
by Starta Institute

ПЛАН ЗАНЯТИЯ

1. Повторение
2. Основной блок
3. Вопросы по основному блоку
4. Домашняя работа

TEL-RAN
by Starta Institute

1 ОСНОВНОЙ БЛОК

Введение

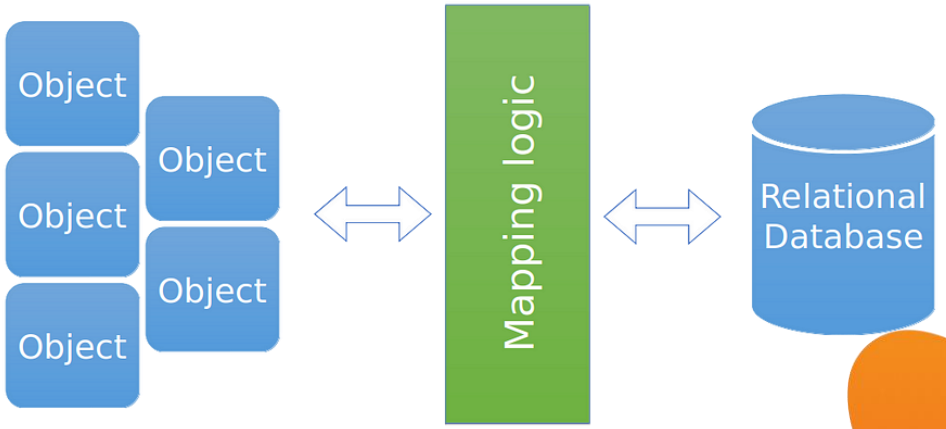
TEL-RAN
by Starta Institute

- Настойчивость
- Сущностный класс
- Выясним отношения
- Жизнь замечательных сущностей
- Что первично?
- Под Data

Проблема

JDBC является частью *JDK* (а не *JEE*) и появилась ещё в *Java 1.1*. Код, который приходится писать в мапперах даже при использовании *JdbcTemplate* является громоздким и однотипным.

Если уж Spring умеет самостоятельно создавать бины из тела http-запроса, то, вероятно, стоит применить данный подход к созданию сущностей из таблиц, полученных по запросу из БД?



Настойчивость

Java Persistence API

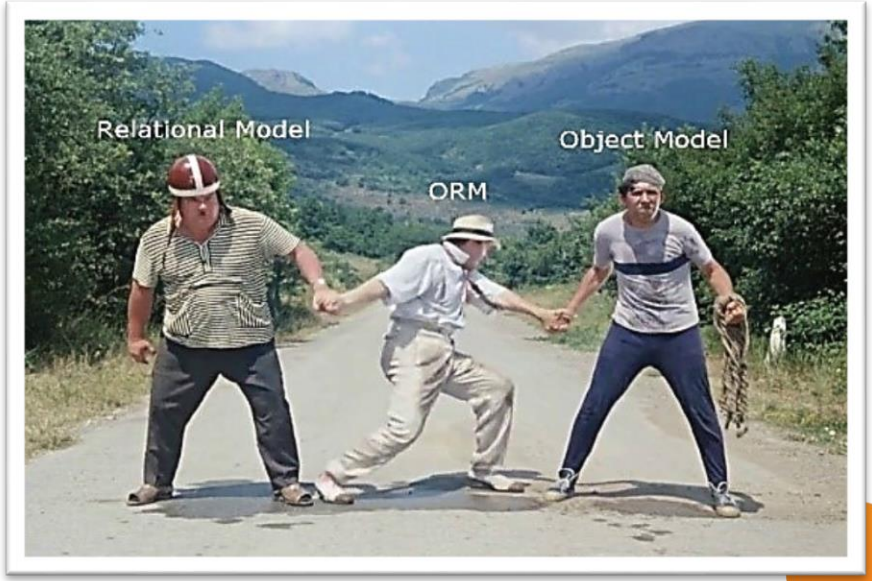
JPA (*Java Persistence API*) – спецификация *JEE*, которая предоставляет стандартизированный способ управления реляционными данными в приложениях *Java*, абстрагирует работу с базой данных, позволяя разработчику взаимодействовать с объектами *Java*, а не с SQL-запросами. Это достигается за счет использования **ORM** (*Object-Relational Mapping*) для маппинга объектов Java в таблицы базы данных и наоборот.

Hibernate - одна из наиболее популярных реализаций JPA, фреймворк для *ORM*, упрощающий работу с JPA в приложениях.

Настойчивость

Object-Relational Mapping

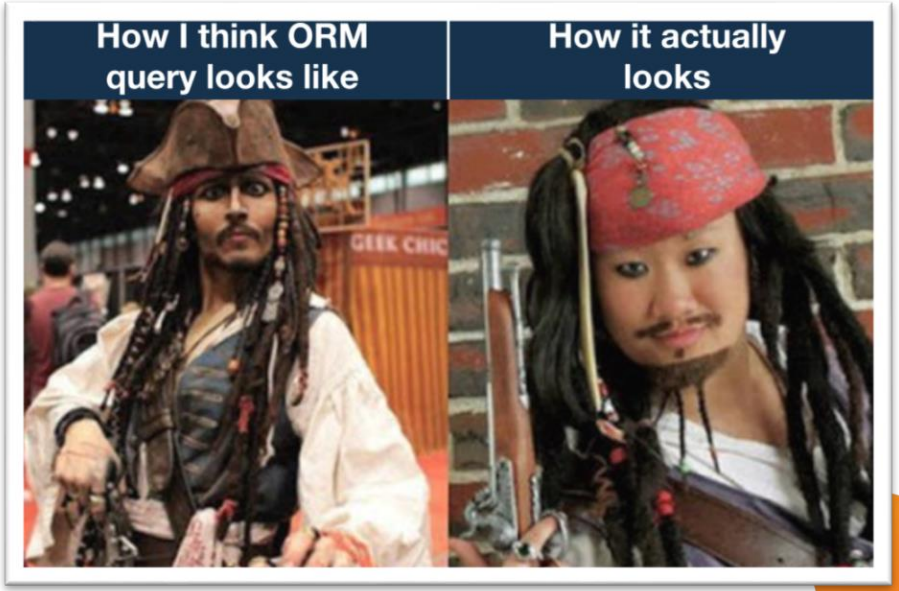
Из-за того, что подходы ООП и реляционных БД принципиально отличаются приходится идти на некоторые ограничения и компромиссы. *ORM* позволяют достигать таких компромиссов и автоматизируют процесс взаимодействия приложения с БД.



Настойчивость

Object-Relational Mapping

Автоматизация, заложенная в *ORM,* обычно позволяет не писать запросы напрямую в БД, а использовать специальные объектно-ориентированные диалекты. Ответственность за формирование конечного запроса ложится на ORM, что, с одной стороны, упрощает код, но с другой не даёт гибкого контроля создаваемых запросов.



Настойчивость

Java Persistence API

JPA, являясь спецификацией, устанавливает некоторые правила, которые с одной стороны ограничивают возможности приложения стандартным набором взаимодействий с БД, а с другой – обеспечивают единство маппинга сущностей ООП и БД. «Под капотом» реализации *JPA* используют *JDBC*.

Сущностный класс

Классы сущностей

Сущность (*entity*) – класс, в объекты которого JPA маппит данные из таблиц БД.

Сущность представляет собой POJO-класс с набором требований и ограничений.

Сущностный класс

Требования JPA к entity-классу

- должен иметь конструктор без аргументов, с *public* или *protected* модификатором;
- может иметь другие конструкторы;
- не должен быть *final*;
- Не должен иметь каких-либо *final*-методов или final-полей экземпляра класса, участвующих в маппинге;
- должен соблюдать инкапсуляцию и сокрытие (доступ через геттеры и сеттеры)
- должен быть отмечен аннотацией *@Entity* или описан в *XML* файле конфигурации *JPA*;
- должен быть классом верхнего уровня (top-level class);
- не может быть *enum* или интерфейсом,
- если объект *entity* класса будет передаваться через удаленный интерфейс, он так же должен реализовывать *Serializable*;
- должен содержать первичный ключ, то есть атрибут или группу атрибутов, которые уникально определяют запись этого Enity-класса в базе данных.

Сущностный класс

Аннотации сущностей

@Entity – аннотация, помечающая POJO-класс как сущность.

@Table – указывает, с какой таблицей связан класс. Может быть опущена, тогда имя таблицы будет взято по имени класса. Параметры аннотации:

- *name* – имя таблицы в БД.
- *catalog* – каталог, в котором определена таблица.
- *schema* – схема, в которой определена таблица.
- *uniqueConstraints* – уникальные ограничения генерации таблицы.
- *indexes* – индексы таблицы, создаваемые при генерации таблицы.

Сущностный класс

Аннотации entity

@Column – аннотация, помечающая связь поля со столбцом таблицы. Может быть опущена, тогда имя столбца формируется по имени поля. Параметры аннотации:

- *name* – имя столбца. По умолчанию формируется из имени поля.
- *nullable* – ограничение на возможность хранить null-значение.
- *insertable* – указывает, включается ли поле в INSERT-запросы (или генерируется).
- *updatable* – указывает, включается ли поле в UPDATE-запросы (или генерируется).
- *table* – имя таблицы, содержащей столбец. По умолчанию подразумевается, что столбец принадлежит главной таблице.
- *length* – ограничение на длину хранимой строки.
- *unique* – указывает ограничение на уникальность хранимых значений.
- *columnDefinition* – фрагмент SQLзапроса, используемый для генерации DDL для столбца.

@Id – аннотация, помечающая поле со столбцом, хранящим простой первичный ключ (числовой или строковый).

Сущностный класс

Аннотации entity

@GeneratedValue – используется в паре с *@Id*, чтобы указать способ генерации *PK*:

generator – имя генератора в БД, *strategy* – стратегия генерации *PK:*

- *GenerationType.IDENTITY* - используется встроенный в БД тип данных столбца *-identity* - для генерации значения *PK*.
- *GenerationType.SEQUENCE* – используется последовательность – специальный объект БД для генерации уникальных значений.
- *GenerationType.TABLE* – для генерации уникального значения используется отдельная таблица, которая эмулирует последовательность. Когда требуется новое значение, *Hibernate* блокирует строку таблицы, обновляет хранящееся там значение и возвращает его обратно в приложение. Эта стратегия наихудшая по производительности и ее желательно избегать.
- *GenerationType.AUTO* – используется по умолчанию. *Hibernate* сначала попробует использовать стратегию *SEQUENCE*, но если БД её не поддерживает (как *MySQL*, например), то будет использована стратегия *TABLE*.

Сущностный класс

Пример entity

```sql
CREATE TABLE users(
    id BIGINT PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
    first_name VARCHAR(100) NOT NULL CHECK (first_name <> ''),
    last_name VARCHAR(100) NOT NULL CHECK (last_name <> ''),
    birth_date DATE,
    email VARCHAR(50)
);

INSERT INTO users (first_name, last_name, birth_date, email)
values ('Joel', 'Edgerton', '1974-06-23', NULL);
INSERT INTO users (first_name, last_name, birth_date, email)
values ('Tom', 'Hardy', '1979-09-15', 'joelEd@gmail.com');
INSERT INTO users (first_name, last_name, birth_date, email)
values ('Nicholas', 'Nolte', '1941-02-08', NULL);
```

```java
@AllArgsConstructor
@NoArgsConstructor
@Getter
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "birth_date")
    private LocalDate birthDate;

    @Column(name = "email")
    private String email;
}
```

Задание

1 Добавьте зависимости

https://mvnrepository.com/artifact/org.springframework/spring-orm

https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-core

2 Доработайте entity, соответствующие ранее созданным таблицам.

Проблема

TEL-RAN
by Starta Institute

В ООП классы могут зависеть друг от друга различными способами – агрегация, композиция, наследование. В реляционных БД взаимоотношения выстраиваются иначе.

Каким образом мы можем соотнести взаимосвязи ООП и реляционных БД?

Выясним отношения

Виды отношений

Реляционные БД поддерживают три основных вида взаимоотношений таблиц:

one-to-one – одной записи таблицы 1 соответствует одна запись таблицы 2. Например, у каждого пользователя есть один документ, удостоверяющий личность.

one-to-many – одной записи таблицы 1 соответствует несколько записей таблицы 2. Например, у одного пользователя есть несколько адресов для доставки товара.

many-to-one – множеству записей таблицы 1 соответствует одна запись таблицы 2. Обратное отношение к *one-to-many*.

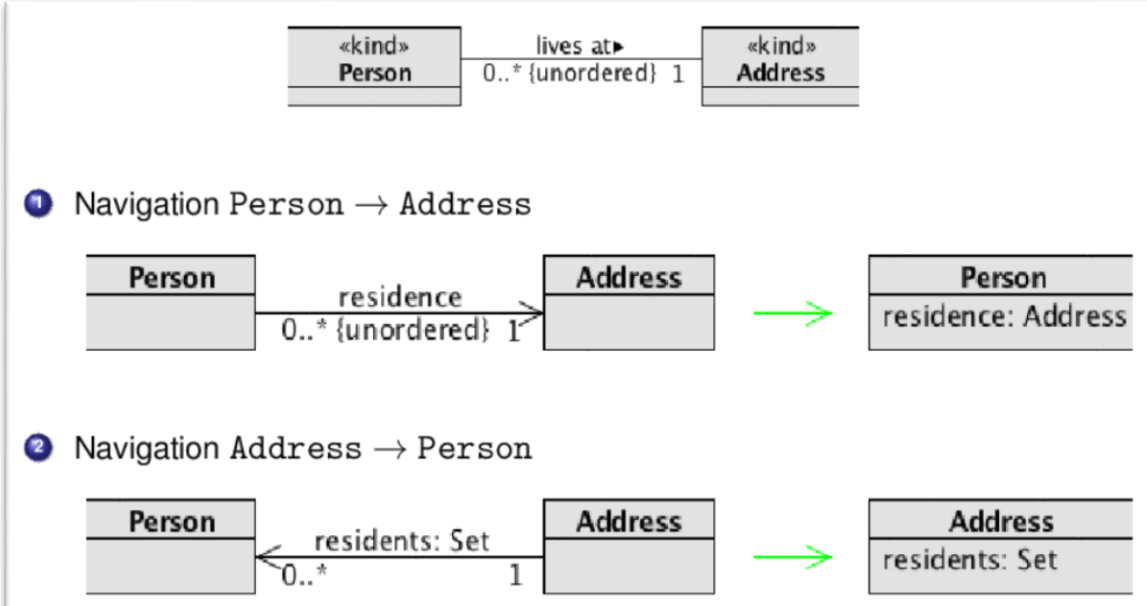
many-to-many – множеству записей таблицы 1 соответствует множество записей таблицы 2. Пользователь заказал много товаров, каждый товар был заказан многими пользователями.

Выясним отношения

Виды отношений

Кроме того, связи могут быть двунаправленными (**bidirectional**) или однонаправленными (**unidirectional**). Если одна таблица ссылается на другую, то это однонаправленная связь. Если вторая в свою очередь также ссылается на первую, то связь двунаправленная. Это влияет на то, как можно достигать нужных данных с помощью запросов.

Например, отношения Штат – Город (один ко многим). Если только штат «знает» о своих городах, то связь однонаправленная. Если города тоже хранят id своего штата (знают, к какому штату относятся), то связь двунаправленная.



Выясним отношения

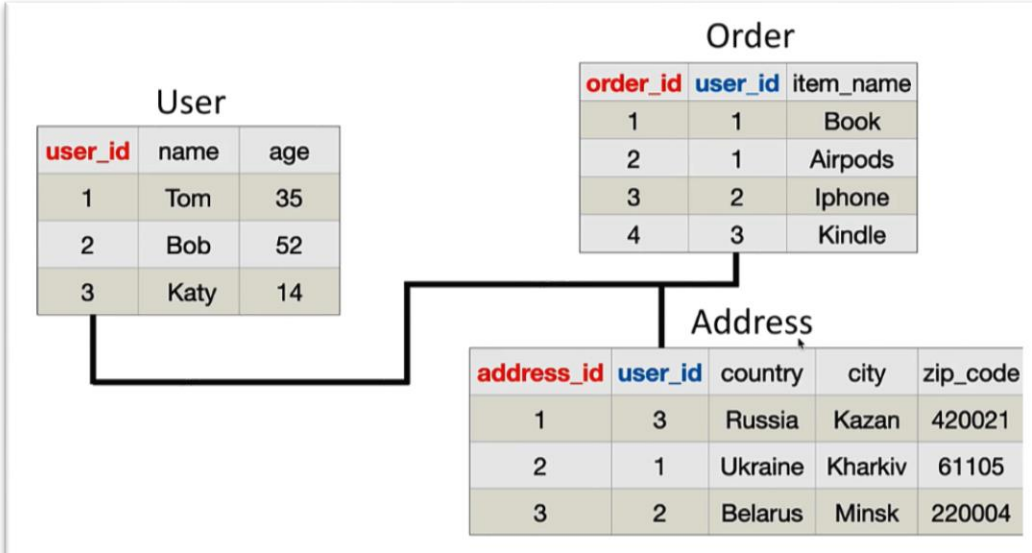
Первичный ключ

Для взаимосвязи разных таблиц в реляционных БД используются *первичные* и *внешние* ключи.

Primary key (*PK*) – первичный ключ, устанавливающий уникальное не-null значение идентификатора для каждой записи в таблице. Хранится в отдельном столбце таблицы.

Foreign key (*FK*) – внешний ключ, который указывает на первичный ключ другой таблицы. Так выполняется связь записей в разных таблицах. Хранится в отдельном столбце таблицы. Значение не может быть null, но может быть неуникально.

В отношении *one-to-one PK* и *FK* могут совпадать и храниться в одном столбце.



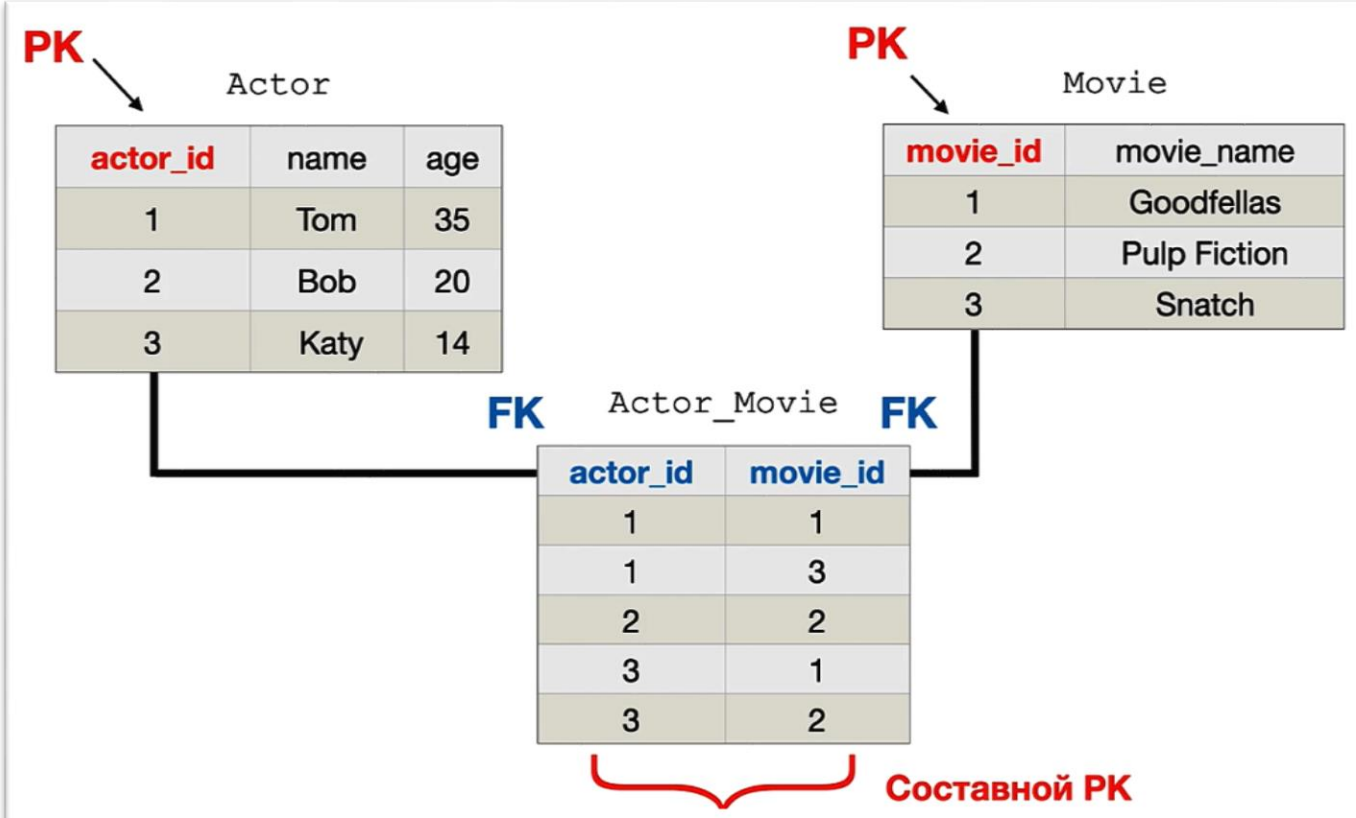
Выясним отношения

Первичный ключ

Выясним отношения

Первичный ключ в many-to-many

Для построения отношения many-to-many используется соединяющая таблица (**Join table**), которая хранит связи *PK* первой таблицы с *PK* второй таблицы. Сама Join table имеет *составной PK*, состоящий из конкатенации FK связываемых таблиц.

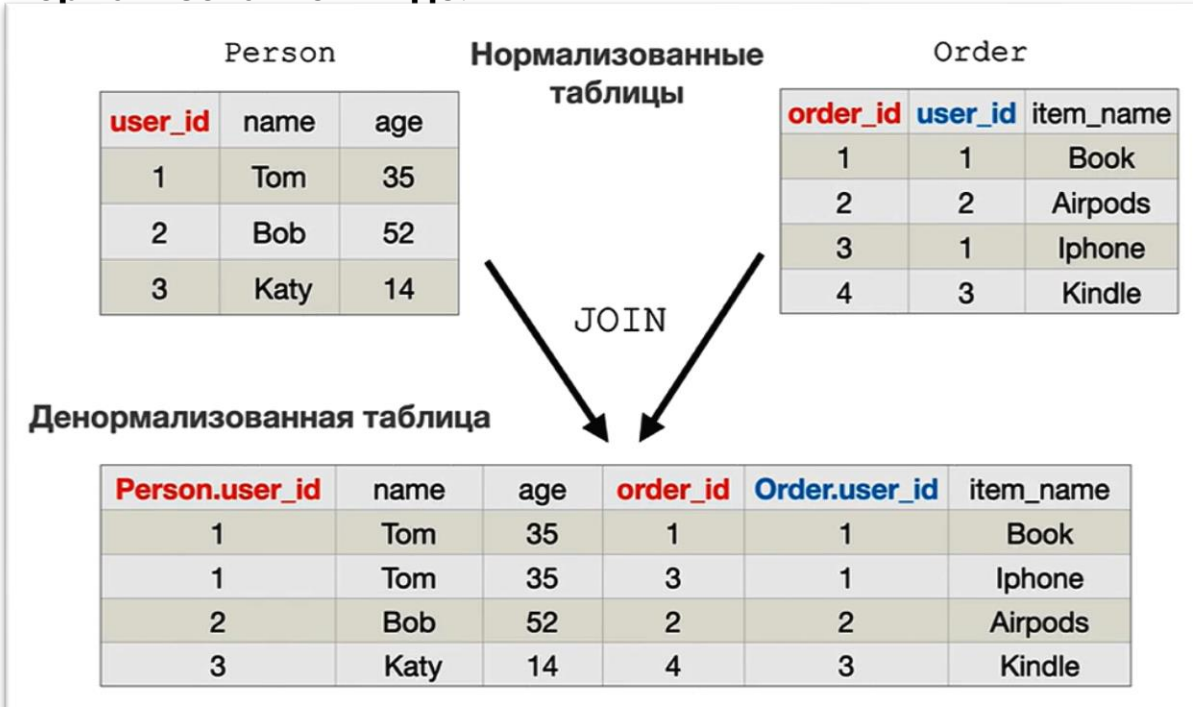


Выясним отношения

Нормализация и объединение

Чтобы данные в БД не дублировались, мы храним их только в одном месте – в одной таблице, в одной записи и даём уникальный идентификатор этой записи. Такой подход называется **хранением данных в нормализованном виде**.

Но когда мы хотим получить данные, то их приходится собирать воедино в одну временную денормализованную таблицу (с дубликатами). Это объединение выполняется с помощью команд *JOIN* в *SQL*.



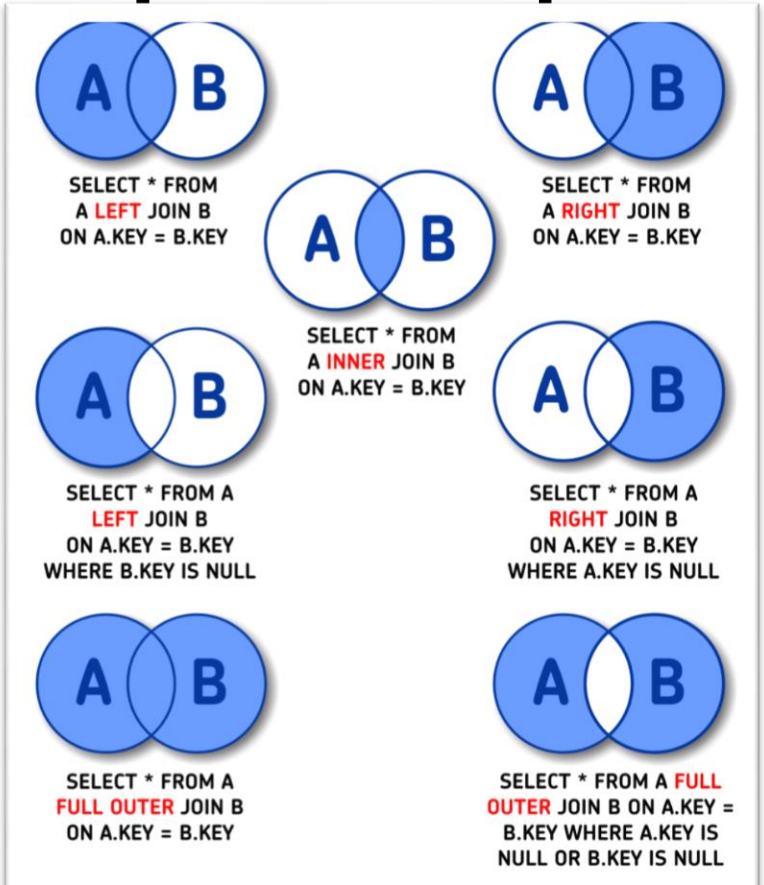
user_id	name	age
1	Tom	35
2	Bob	52
3	Katy	14

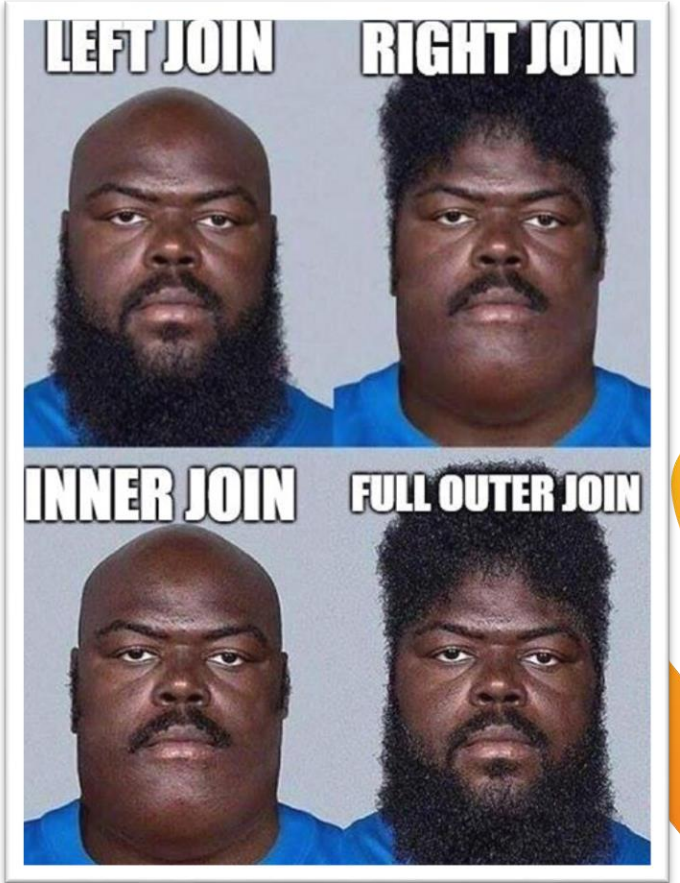
order_id	user_id	item_name
1	1	Book
2	2	Airpods
3	1	Iphone
4	3	Kindle

Person.user_id	name	age	order_id	Order.user_id	item_name
1	Tom	35	1	1	Book
1	Tom	35	3	1	Iphone
2	Bob	52	2	2	Airpods
3	Katy	14	4	3	Kindle

Выясним отношения

Нормализация и объединение





Выясним отношения

Отношение one-to-one

Существует несколько способов выстроить отношение one-to-one в JPA.

1 Аннотация **@OneToOne** кажется наиболее логичным способом организации связи двух объектов.

```
public enum ConfirmationMethod {
    EMAIL,
    PHONE,
    GOOGLE_ACCOUNT,
    FACEBOOK_ACCOUNT
}
```

Такой подход создаёт в таблице REGISTRATION внешний ключ USER_ID, указывающий на ID в таблице USER

```
@Data
@NoArgsConstructor
@Entity
@Table(name = "registrations")
public class Registration {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private long id;
    private LocalDate date;
    @Enumerated(EnumType.STRING)
    private ConfirmationMethod confirmationMethod;
    @OneToOne(fetch = FetchType.LAZY)
    private User user;
}
```

Выясним отношения

Отношение one-to-one

При этом класс *User* тоже может ссылаться на *Registration* (или не ссылаться, на схему это не влияет). Однако такой подход имеет недостатки:

- лишний столбец
- если *User* тоже в свою очередь ссылается на Registration, то его настройка *fetch = FetchType.LAZY* не работает. То есть при поиске пользователя генерируется не один, а два SQL запроса.

```
@AllArgsConstructor @NoArgsConstructor @Getter
@Entity @Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id") private Long id;

    @Column(name = "first_name") private String firstName;

    @Column(name = "last_name") private String lastName;

    @Column(name = "birth_date") private LocalDate birthDate;

    @Column(name = "email") private String email;

    @OneToOne(
                mappedBy = "user",
                cascade = CascadeType.ALL,
                optional = false,
                fetch = FetchType.LAZY
    )
    private Registration registration;
}
```

Выясним отношения

Отношение one-to-one

При этом класс *User* тоже может ссылаться на *Registration* (или не ссылаться, на схему это не влияет). Однако такой подход имеет недостатки:

- лишний столбец
- если *User* тоже в свою очередь ссылается на Registration, то его настройка *fetch = FetchType.LAZY* не работает. То есть при поиске пользователя генерируется не один, а два SQL запроса.

```
@AllArgsConstructor @NoArgsConstructor @Getter
@Entity @Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id") private Long id;

    @Column(name = "first_name") private String firstName;

    @Column(name = "last_name") private String lastName;

    @Column(name = "birth_date") private LocalDate birthDate;

    @Column(name = "email") private String email;

    @OneToOne(
            mappedBy = "user",
            cascade = CascadeType.ALL,
            optional = false,
            fetch = FetchType.LAZY
    )
    private Registration registration;
}
```

У отношений в JPA всегда есть владеющая сторона и обратная. Обратная указывает настройку mappedBy

Выясним отношения

Отношение one-to-one

Установление связей двустороннего one-to-one

```
User userOne = session.find(User.class, 1);
Registration registration = session.find(Registration.class, 101);
registration.user = userOne;
userOne.registration = registration;

session.update(registration);
session.flush();
```

Для удаления связи ссылки тоже нужно удалить у обоих объектов:

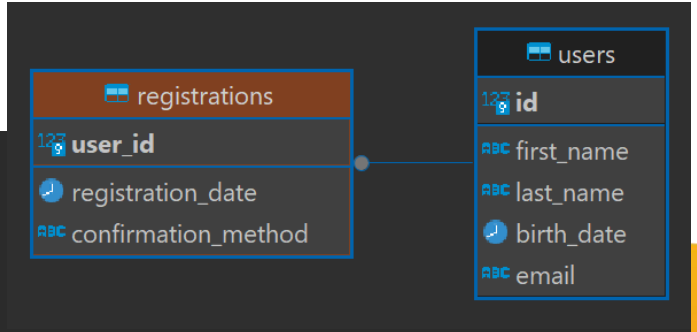
```
User userOne = session.find(User.class, 1);
Registration registration = userOne.registration;

registration.user = null;
session.update(registration);

userOne.registration = null;
session.update(userOne);

session.flush();
```

Выясним отношения

Отношение one-to-one

2 Более оптимальный способ с *@OneToOne*.

Если у каждого объекта *Registration* свой ровно один *User*, то в таблице REGISTRATIONS не нужен автогенерируемый *PK*. Достаточно одного USER_ID – пусть он будет *PK* и *FK* одновременно.




```
CREATE TABLE registrations(
                    registration_date DATE,
                    confirmation_method VARCHAR(50),
                    user_id BIGINT NOT NULL,
                    CONSTRAINT registration_pkey PRIMARY KEY (user_id),
                    CONSTRAINT fk_registrations_to_user FOREIGN KEY (user_id)
                        REFERENCES users (id) MATCH SIMPLE
                        ON UPDATE NO ACTION
                        ON DELETE NO ACTION);
INSERT INTO registrations (user_id, registration_date, confirmation_method) values (1, '2024-02-22', 'EMAIL');
INSERT INTO registrations (user_id, registration_date, confirmation_method) values (2, '2023-01-11', 'EMAIL');
INSERT INTO registrations (user_id, registration_date, confirmation_method) values (3, '2022-12-12', 'PHONE');
```

Выясним отношения

Отношение one-to-one

Аннотация **@MapsId** внедряет в поле *@Id* текущей сущности значение *PK* из родительской сущности. Т.е. id не генерируется автоматически, а заполняется идентификатором *User*

Помимо более чистой структуры БД, для поля *registration* сущности *User* начинает работать *FetchType.LAZY*, то есть при поиске *User* по *id* уже выполняется один *select*, а не два. Тем не менее, в таких случаях не стоит делать обратную ссылку из *User* в *Registration* (двунаправленная связь), т.к. зная идентификатор *User*, всегда можно извлечь *Registration* по такому же идентификатору. И уж тогда получить второй *select*.

```
@Data
@NoArgsConstructor
@Entity
@Table(name = "registrations")
public class Registration {
    @Id
    @Column(name = "user_id")
    private long userId;
    private LocalDate registrationDate;
    @Enumerated(EnumType.STRING)
    private ConfirmationMethod confirmationMethod;
    @OneToOne(fetch = FetchType.LAZY)
    @MapsId
    private User user;
}
```

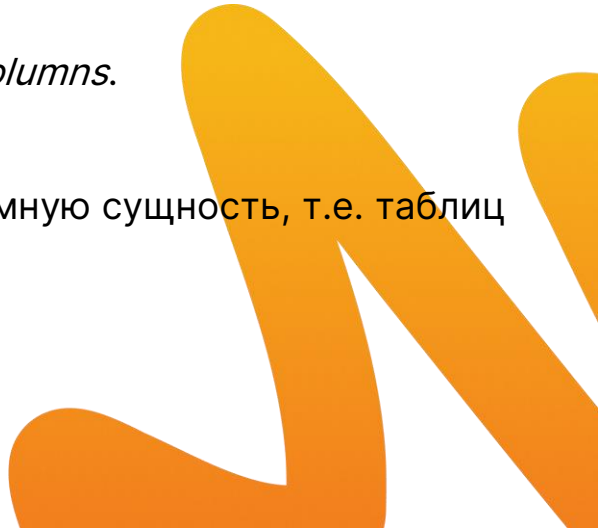
Выясним отношения

Отношение one-to-one

3 Помимо аннотации *@Table*, связывающей сущность с таблицей БД, можно указать аннотацию **@SecondaryTable** или **@SecondaryTables** для связывания отдельных полей сущности со второй и последующей таблицами. *@SecondaryTable* имеет те же параметры, что и *@Table*, плюс:

- *pkJoinColumns* – определяет, по какому столбцу соединять вторичную и первичную таблицы.
- *foreignKey* – определяет *FK,* связанный с элементом *pkJoinColumns*.

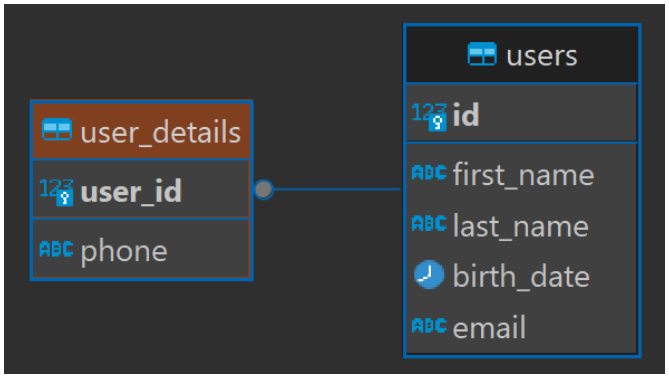
@SecondaryTable объединяет несколько таблиц в одну программную сущность, т.е. таблиц много, а класс с *@Entity* один.



Выясним отношения

Отношение one-to-one

Пример использования **@SecondaryTable**




```java
@AllArgsConstructor @NoArgsConstructor @Getter
@Entity
@Table(name = "users")
@SecondaryTable(name = "user_details",
    pkJoinColumns = @PrimaryKeyJoinColumn(name = "user_id"))
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "birth_date")
    private LocalDate birthDate;

    @Column(name = "email")
    private String email;

    @Column(name="phone", table="user_details")
    private String phone;
}
```

Выясним отношения

Отношение one-to-one

4 Указанные выше методы требуют создания отдельной таблицы в БД, но данные собираются в одну сущность. Для обратной ситуации, когда два класса хранятся в одной таблице в БД используются аннотации **@Embeddable** или **@Embedded.**

```
@Data
@NoArgsConstructor
@Embeddable
public class Settings {
    @Column(name = "is_dark_theme", columnDefinition = "boolean default false")
    private boolean isDarkTheme;
    @Column(name = "is_save_history", columnDefinition = "boolean default false")
    private boolean isSaveHistory;
    @Column(name = "nickname")
    private String nickname;
}
```

Сущность с *@Embeddable* может хранить поля составного *PK.* Тогда вместо *@Embedded* нужно использовать **@EmbeddedId**.

```
public class User {
                …
    @Embedded
    private Settings settings;
}
```

Задание

1 Создайте таблицу, хранящую документ, удостоверяющий личность (паспорт), для пользователей. Свяжите таблицу паспортов с таблицей пользователей. У паспорта должны быть поля id, буквенная или числовая серия и номер. Создайте паспорта для двух пользователей. Выполните запрос на получение информации о пользователях вместе с данными паспортов.

2 Свяжите сущности Passport и User.

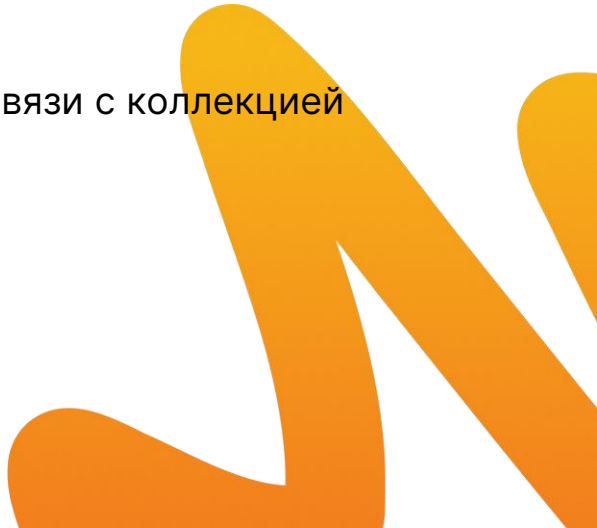
Выясним отношения

Отношения one-to-many и many-to-one

Использование односторонней связи сущностей по *@OneToMany* приводит к появлению дополнительных UPDATE-запросов при каждом INSERT-запросе, поэтому аннотации **@OneToMany** и **@ManyToOne** всегда используют в паре, образуя двустороннюю связь.

Другой способ образовать связь одной сущности с набором других – это аннотации

- **@ElementCollection** – определяет коллекцию сущностей базовых типов (примитивные, String, Date) или сущностей с *@Embeddable.*
- **@CollectionTable** – определяет таблицу, используемую для связи с коллекцией элементов базового типа или сущностей с *@Embeddable.*



Выясним отношения

Отношения one-to-many и many-to-one

```java
@Data
@NoArgsConstructor
@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "order_date")
    private LocalDate orderDate;

    @Column(name = "items")
    private List<GoodsItem> items;

    @ManyToOne(fetch = FetchType.LAZY)
    private User user;
}
```

```java
public class User {
    …
    @OneToMany(
        mappedBy = "user",
        cascade = CascadeType.ALL,
        orphanRemoval = true
    )
    private List<Order> orders = new ArrayList<>();

    public void addOrder(Order order) {
        orders.add(order);
        order.setUser(this);
    }
    public void removeOrder(Order order) {
        orders.remove(order);
        order.setUser(null);
    }
}
```

Задание

Создайте класс Comments, хранящий комментарии пользователя о товаре.

Предполагается, что на каждый товар пользователь оставляет не более одного комментария.

Выясним отношения

Отношение many-to-many

Для создания отношения коллекция – коллекция используется аннотация **@ManyToMany** надо соответствующими полями сущностей. Поле сущности может быть выражено коллекцией List или Set. Но применять рекомендуется именно Set, т.к. при удалении элемента из List сначала удаляются все записи для заданного значения, а потом обратно вставляются те, что удалять не надо, что не оптимально.

Выясним отношения

Отношение many-to-many

```java
@Data
@NoArgsConstructor
@Entity
@Table(name = "goods_items")
public class GoodsItem {
    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "item_name")
    private String itemName;

    @Column(name = "description")
    private String description;

    @Column(name = "price")
    private BigDecimal price;

    @ManyToMany(mappedBy = "items")
    private Set<Order> orders = new HashSet<>();
```

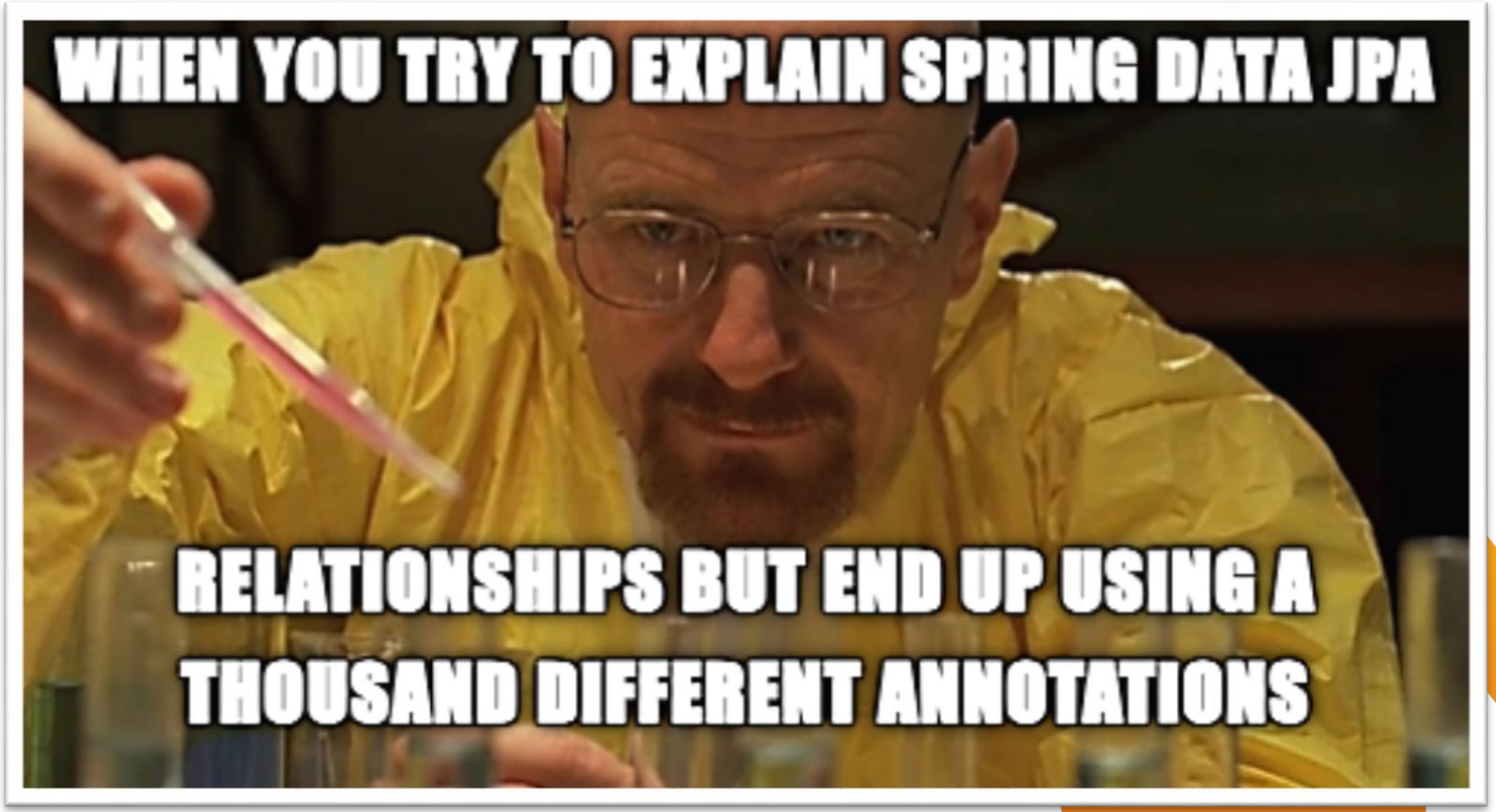
```java
public class Order {
    …
    @ManyToMany (cascade = {
            CascadeType.PERSIST,
            CascadeType.MERGE
    })
    @JoinTable(name = "order_item",
            joinColumns = @JoinColumn(name = "order_id"),
            inverseJoinColumns = @JoinColumn(name = "item_id")
    )
    private List<GoodsItem> items = new ArrayList<>();

    public void addItem(GoodsItem item){
        this.items.add(item);
        item.getOrders().add(this);
    }
    public void removeItem(GoodsItem book){
        this.items.remove(book);
        book.getOrders().remove(this);
    }
}
```

Задание

Предположим, бизнес проводит периодические вебинары по продвижению товаров. Создайте класс Event, хранящий информацию о событии, которое посетил пользователь (дата, название). Один пользователь может посетить многие события и одно событие могут посетить многие пользователи.

Выясним отношения

Слишком много аннотаций

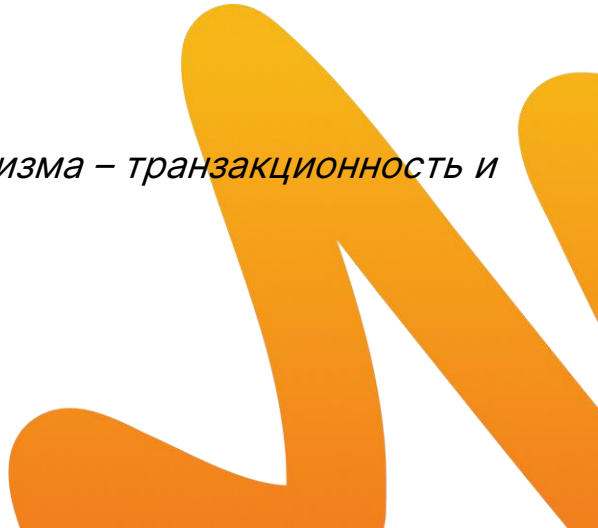


Проблема

TEL-RAN
by Starta Institute

ORM представляет собой прослойку между БД и приложением. Из-за наличия оптимизаций доступа к БД и кэшей, которые ускоряют работу приложения с БД, разработчик приложения теряет возможность прямого доступа к БД. Т.е. появляется состояние ~~беспомощности и гнева у разработчика~~ сущности, при котором приложение уже считает сущность изменённой/удалённой/прочитанной, но в реальности запрос к БД ещё не выполнен.

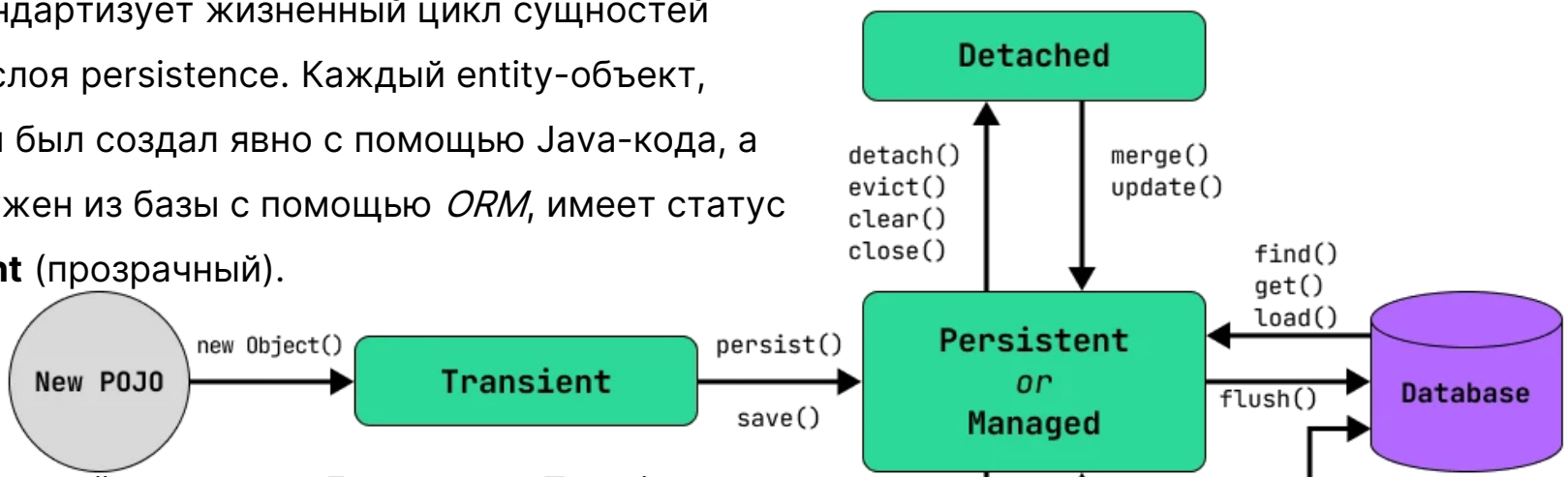
От расхождения данных приложения и БД защищают два механизма – транзакционность и понимание жизненного цикла сущности.



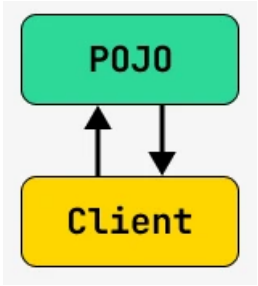
Жизнь замечательных сущностей

Transient

JPA стандартизует жизненный цикл сущностей (entity) слоя persistence. Каждый entity-объект, который был создал явно с помощью Java-кода, а не загружен из базы с помощью *ORM*, имеет статус **Transient** (прозрачный).



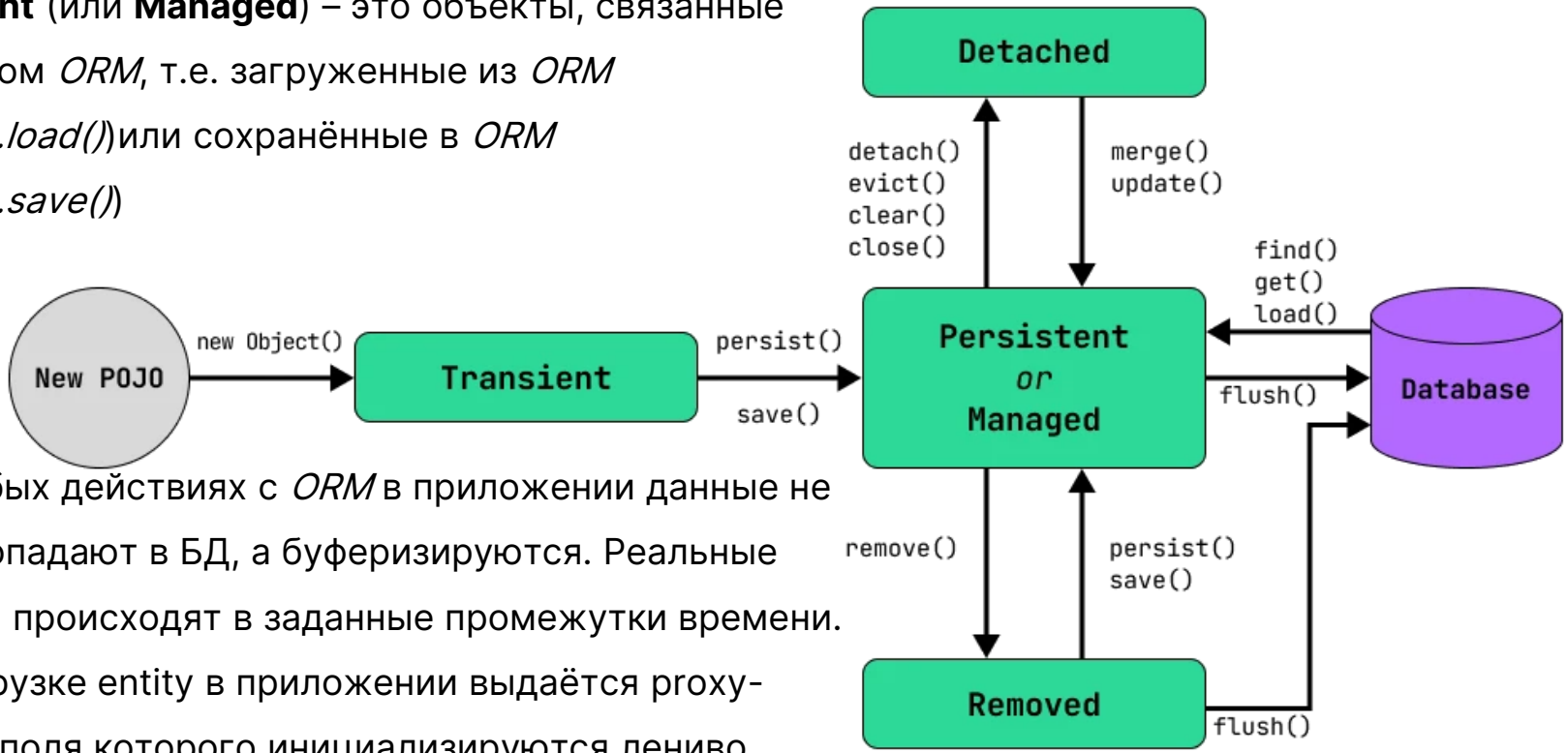
ORM пока о нём не знает. Диаграмма *Transient*-объекта:



Жизнь замечательных сущностей

Persistent

Persistent (или **Managed**) – это объекты, связанные с движком *ORM*, т.е. загруженные из *ORM* (*session.load()*)или сохранённые в *ORM* (*session.save()*)



При любых действиях с *ORM* в приложении данные не сразу попадают в БД, а буферизируются. Реальные запросы происходят в заданные промежутки времени. При загрузке entity в приложении выдаётся proxy-объект, поля которого инициализируются лениво.

Жизнь замечательных сущностей

Persistent

Такому объекту обычно соответствует какая-то запись в базе данных, у него есть идентификатор и такой объект присоединен к сессии *ORM*

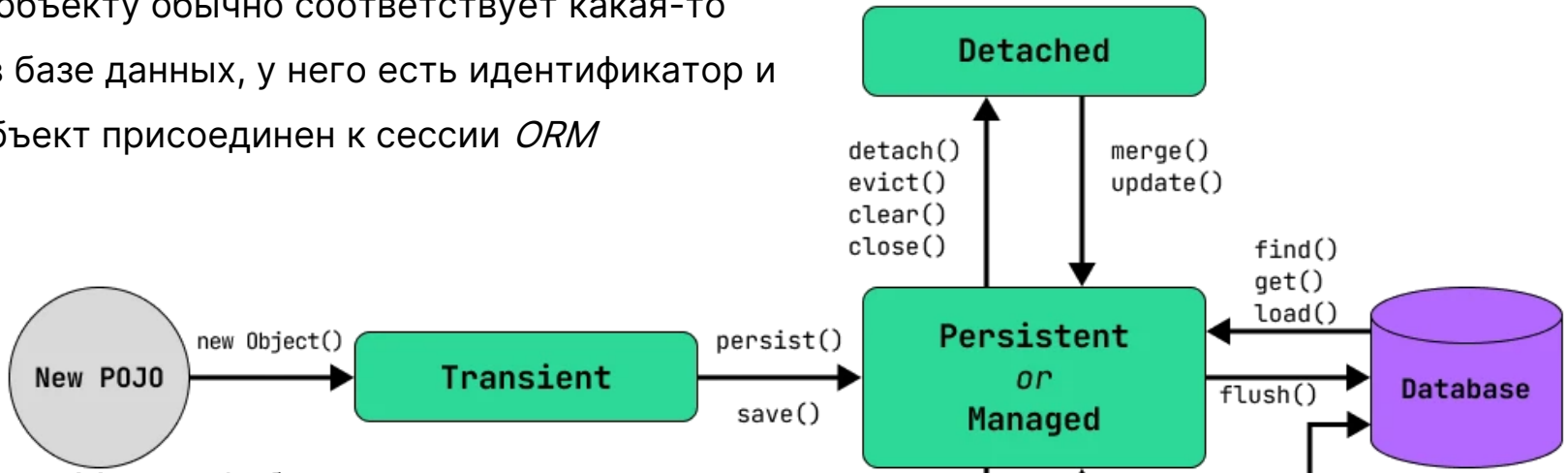
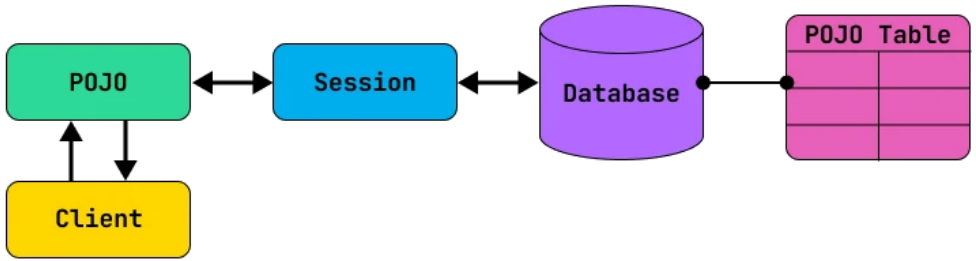


Диаграмма *Managed*-объекта:



Жизнь замечательных сущностей

Detached

Detached – состояние, при котором объект был отсоединен от сессии, т.е. сессия закрылась (*session.close()*) или транзакция завершилась (*session.evict(entity)*), и *ORM* больше не следит за этим объектом.

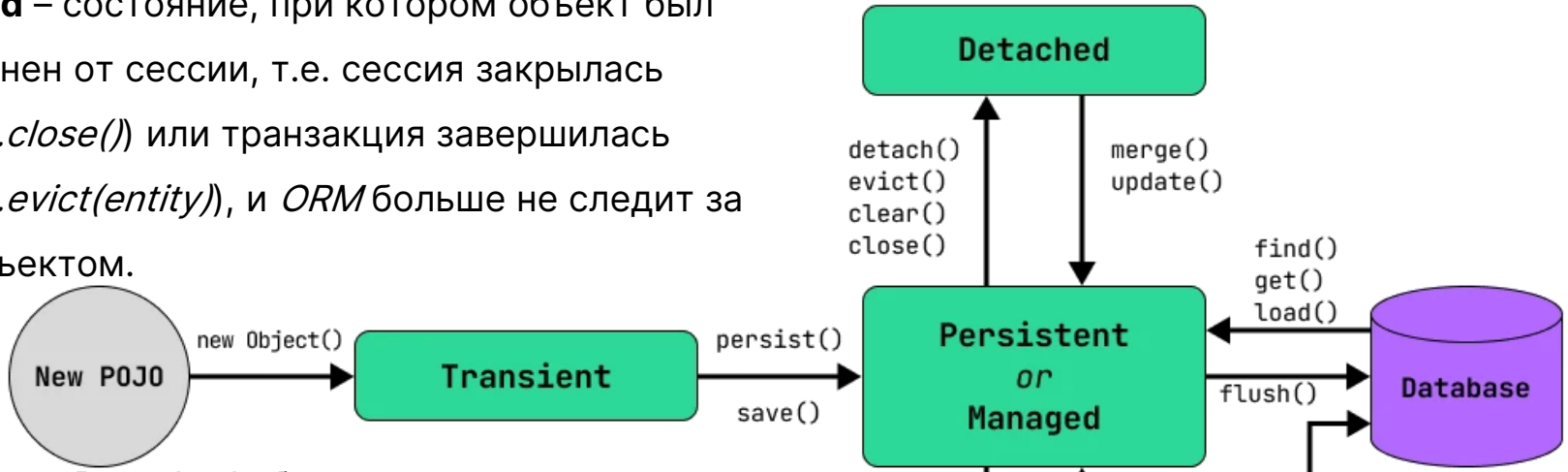
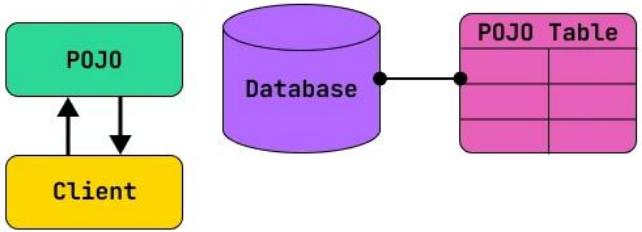


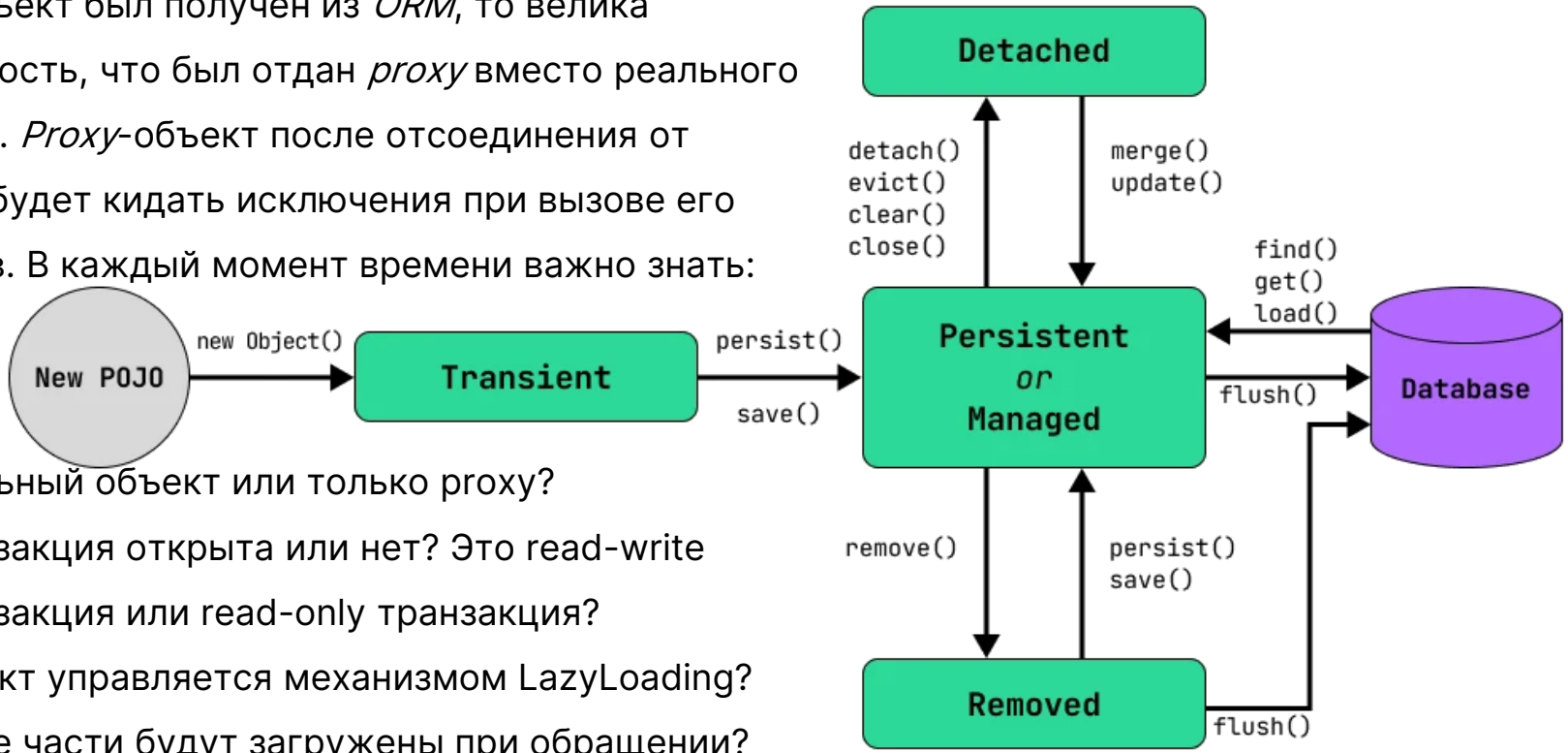
Диаграмма *Detached*-объекта:



Жизнь замечательных сущностей

Detached

Если объект был получен из *ORM*, то велика вероятность, что был отдан *proxy* вместо реального объекта. *Proxy*-объект после отсоединения от сессии будет кидать исключения при вызове его методов. В каждый момент времени важно знать:

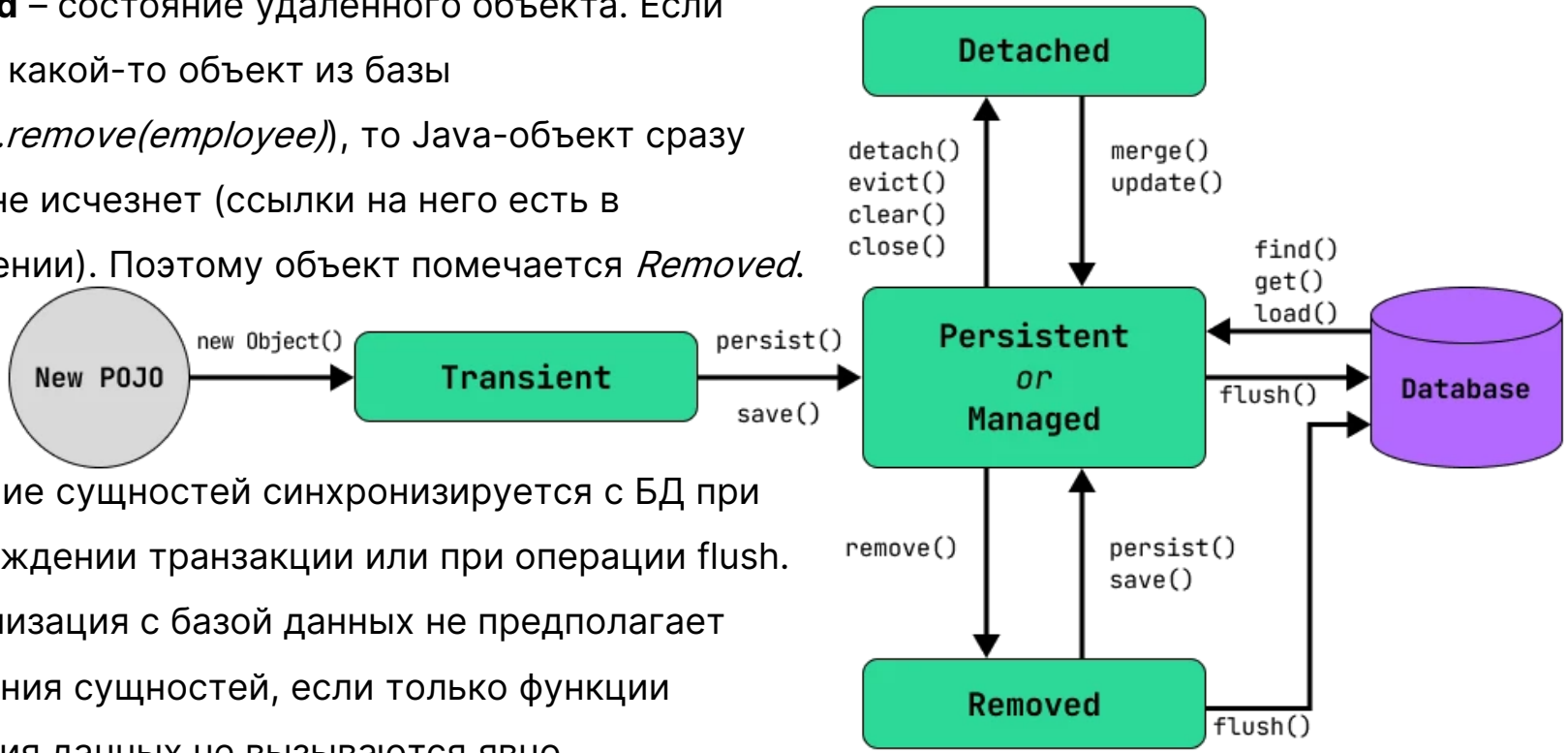


- реальный объект или только proxy?
- транзакция открыта или нет? Это read-write транзакция или read-only транзакция?
- объект управляется механизмом LazyLoading?
- какие части будут загружены при обращении?
- как объект соединен с зависимыми объектами?

Жизнь замечательных сущностей

Removed

Removed – состояние удаленного объекта. Если удалить какой-то объект из базы (*session.remove(employee)*), то Java-объект сразу никуда не исчезнет (ссылки на него есть в приложении). Поэтому объект помечается *Removed*.



Состояние сущностей синхронизируется с БД при подтверждении транзакции или при операции flush. Синхронизация с базой данных не предполагает обновления сущностей, если только функции получения данных не вызываются явно.

Жизнь замечательных сущностей

Пример жизненного цикла entity

```
User user = session.load(User.class, 1); //после загрузки у объекта состояние Persisted

session.remove(user); //после удаления у объекта состояние Removed

session.save(user); //а теперь снова Persisted

session.close(); //а теперь состояние Detached
```

Из примера видно, что в JPA и Hibernate основные действия с сущностями выполняются во время сессии (интерфейс *Session*). Сессия является **транзакционной**, т.е. поделена на простейшие операции. После выполнения операции транзакция закрывается с подтверждением сохранения нового результата в БД. При возникновении ошибки во время транзакции все выполненные действия отменяются (*rollback*), что гарантирует сохранность данных.

Жизнь замечательных сущностей

Entity Manager vs Session

Хотя *Hibernate* является реализацией *JPA*, есть ключевые отличия. Например, мы можем управлять жизненным циклом сущностей с помощью *EntityManager* в *JPA* и *Session* в *Hibernate*.

Жизнь замечательных сущностей

Hibernate Session

Session – *API*, специфичный для *Hibernate.* Основной интерфейс *Hibernate*, через который происходит взаимодействие с базой данных, и предоставляющий функциональность, выходящую за рамки *JPA*. *Session* позволяет делать

- управление вторичным кэшем
- нативные SQL-запросы
- критерии (*Criteria API*)
- разработчик сам управляет открытием и закрытием сессий, что может привести к более тонкому контролю над операциями, но также требует более тщательного управления ресурсами
- управление транзакциями осуществляется непосредственно через API Session.

Жизнь замечательных сущностей

Entity Manager

Entity Manager – часть спецификации JPA, интерфейс, который предоставляет абстрактный слой для управления персистентностью объектов Java. При его использовании код легко переносится и может работать с любой БД, поддерживающей JPA.

Entity Manager помогает управлять контекстом персистентности (в т.ч. кэширование, отслеживание и управление жизненным циклом сущностей). Управление транзакциями обычно делегируется контейнеру (Spring контейнеру в *Spring Framework*).

Жизнь замечательных сущностей

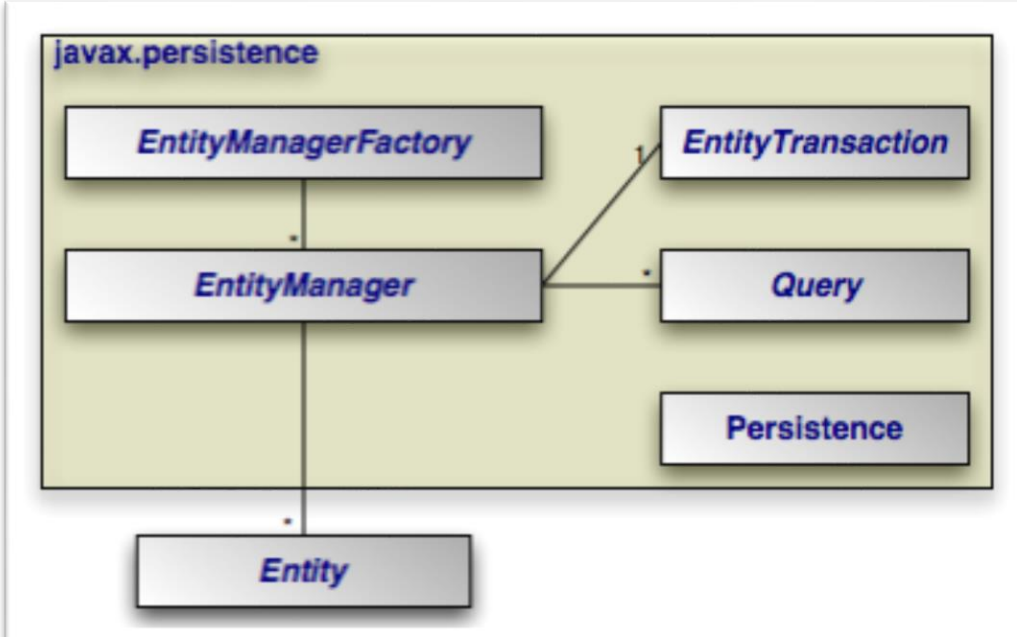
Entity Manager

Контекст персистенстности (*Persistence context*) – контейнер (множество) сущностей, управляющий их жизненным циклом.

Для внедрения *EntityManager* используется аннотация **@PersistenceContext** над полем.

Для внедрения через конструктор придётся явно создавать бин в конфигурации.

Структура основных сущностей *JPA*.



Жизнь замечательных сущностей

Методы Entity Manager

persist() – сделать сущность управляемой и хранимой.

merge() – обновляет состояние сущности в БД, синхронизируя текущее состояние переданного объекта сущности с тем, что хранится в БД. Если сущность не управляется в текущий момент (т.е., она является *detached*), то *merge()* копирует состояние в управляемую (*managed*) сущность и возвращает эту управляемую версию.

remove() – удалить сущность.

find() – найти сущность (сначала в контексте, затем в БД) по *PK*.

lock() – заблокировать сущность в контексте определённым режимом блокировки.

refresh() – обновить состояние сущности данными из БД, перезаписав внесённые ранее изменения (если они были).

detach() – удалить сущность из контекста, что приводит к изменению её статуса с *managed* на *detached*. Если не был вызван метод *flush()*, то внесённые в такую сущность изменения (включая пометку к удалению) не будут синхронизированы с БД.

Жизнь замечательных сущностей

Пример Entity Manager

SpringRelationsExample.zip

Проблема

TEL-RAN
by Starta Institute

В отношениях приложения и СУБД часто возникает вопрос о том, что первично – приложение или данные. Технология *JDBC* в большей степени считала первичной БД и просто предоставляла возможность приложению делать запросы в БД. Но *JPA*, напротив, старается сделать приложение с его принципами ООП первичным, а базе данных отводится роль простого хранилища данных, которое легко можно заменить. Так *Hibernate* может не только управлять запросами к данным, но и при необходимости создавать необходимые таблицы в БД при старте приложения.

Каким образом можно настроить нужное поведение Hibernate?

Что первично?

Настройка Hibernate

Для настройки чистого *Hibernate* в папке ресурсов приложения обычно создают конфигурационный файл *hibernate.cfg.xml*.

hibernate.cfg.xml

В не-Spring приложениях данный файл вычитывается в специальном утилитарном классе. Spring-приложение (без *Spring Boot*) подхватывает настройки при инициализации бина *SessionFactory* в конфигурационном классе или *xml*-файле конфигурации *Spring*.

Для работы *JPA* потребуется в конфигурации создать *LocalSessionFactoryBean* или *EntityManagerFactory* – в зависимости от необходимости использовать чистый *JPA* или *Hibernate*.

Что первично?

Настройка Hibernate

LocalSessionFactoryBean используется для конфигурации *SessionFactory*. Метод *setConfigLocation* указывает *Spring*, где находится файл *hibernate.cfg.xml*. Также нужно указать пакеты, содержащие сущности, с помощью метода *setPackagesToScan*.

Бин источника данных *DataSource* мы настраивали ранее в другом файле конфигурации

```
@Configuration
public class HibernateConfig {
    // Метод создания SessionFactory, использующего настройки из hibernate.cfg.xml
    @Bean
    public LocalSessionFactoryBean sessionFactory(DataSource dataSource) {
        LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource);
        // Указываем расположение hibernate.cfg.xml
        sessionFactory.setConfigLocation(new ClassPathResource("hibernate.cfg.xml"));
        // Указываем пакеты, содержащие сущности
        sessionFactory.setPackagesToScan("org.example.entity");
        return sessionFactory;
    }
}
```

Что первично?

Настройка Hibernate

Однако, работу с *xml*-файлом можно заменить на конфигурирование в коде.

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {
    LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();
    em.setDataSource(dataSource);
    em.setPackagesToScan("org.example.entity"); // Указываем пакеты, содержащие сущности
    HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    em.setJpaVendorAdapter(vendorAdapter);
    Properties properties = new Properties();
    properties.setProperty("hibernate.hbm2ddl.auto", "update"); // Настройка ddl-auto
    properties.setProperty("hibernate.dialect", "org.hibernate.dialect.PostgreSQLDialect"); // Настройка диалекта
    properties.setProperty("hibernate.show_sql", "true"); // Настройка отображения запросов
    properties.setProperty("hibernate.format_sql", "true"); // Форматирование запросов или всё в одну строку
    properties.setProperty("hibernate.use_sql_comments", "true"); // Доп. комментарии к запросам
    em.setJpaProperties(properties);
    return em;
}
```

В *Spring Boot* все необходимые настройки автоматически подгружаются из файла *application.properties* или *application.yml*, поэтому даже не требуется создавать *SessionFactory*.

Что первично?

Настройка Hibernate

Значение настройки *hibernate.hbm2ddl.auto* указывает, как *Hibernate* будет вести себя с БД после запуска приложения:

- *create* – создавать схему с нуля при каждом запуске (полезно при начальной разработке приложения, когда схема часто меняется и при тестировании). После завершения работы приложения схема продолжит существовать.
- *create-drop* – то же, но схема удаляется при завершении работы приложения.
- *update* – автоматически создавать таблицы на основе программных сущностей. Если какие-то таблицы существуют, то *Hibernate* не будет их изменять, а только досоздаст недостающие (полезно при доработке приложения).
- *validate* – при старте приложения *Hibernate* проверит, что сущности соответствуют схеме БД. При нахождении различий будет брошено исключение (полезно для приложений, где используются инструменты миграции БД – *Flyway* или *Liquibase*).

Задание

1 Создайте выполните запуск приложения с настройкой *hibernate.hbm2ddl.auto=create-drop*

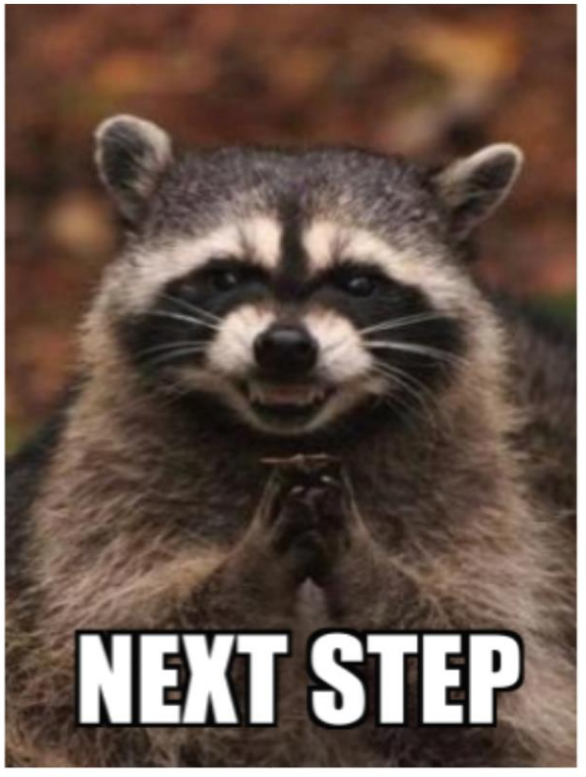
2 Создайте DAO на основе *EntityManager* для ранее добавленных сущностей.

Проблема

TEL-RAN
by Starta Institute

Реализации DAO-классов часто очень похожи, т.к. используется один и тот же принцип – CRUD.

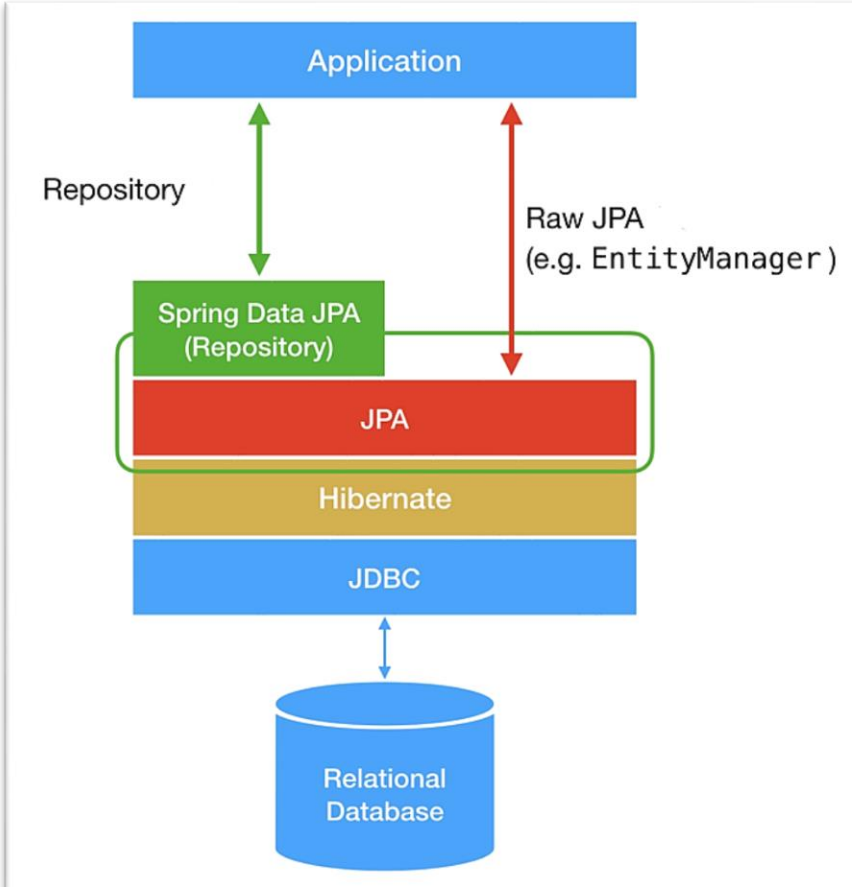
Есть ли вариант автоматизировать и это?



Под Data

Spring Data JPA

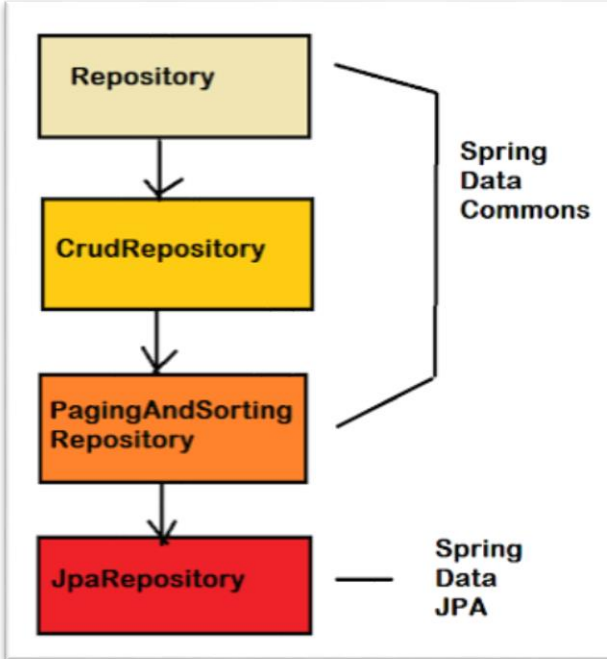
Spring Data JPA – часть экосистемы *Spring*, которая предоставляет более высокоуровневый способ работы с данными через различные хранилища данных, включая реляционные и нереляционные базы данных. *Spring Data* стремится упростить работу с данными, автоматизируя рутинные операции, и предоставляет консистентный способ доступа к данным через различные технологии хранения. В контексте работы с реляционными базами данных, *Spring Data* может использовать *JPA* и *Hibernate* для выполнения операций с данными.



Под Data

Технологии взаимодействия с БД

JpaRepository – это параметризируемый интерфейс, который позволяет не писать реализацию DAO, а создавать репозитории в декларативном стиле. Т.е. объявленные в интерфейсе методы будут реализованы *Spring Data* автоматически, основываясь на данных о параметрах репозитория и названиях методов, представленных в интерфейсе. Достаточно расширить *JpaRepository,* чтобы получить готовый DAO. Интерфейс уже содержит основные CRUD-методы и может быть дополнен нужными методами. Параметрами интерфейса являются тип сущности и тип её *PK*.




```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findAllByLastNameAndFirstNameAndEmail(String lastName, String firstName, String email);
}
```

TEL-RAN
by Starta Institute

2 Домашнее задание

Домашнее задание

TEL-RAN
by Starta Institute

Дополните проект, с которым мы работали на лекции следующими сущностями:

1. UserVerificationData – сущность хранит данные, используемые для верификации пользователя (фотография, запись голоса, биометрические данные, статус верификации). У одного пользователя есть один набор верификационных данных.
2. Статус пользователя UserStatus (поля isVip и grade – SILVER, GOLD, PLATINUM, NONE). У пользователя может быть только один текущий статус. Один тот же статус может быть у нескольких пользователей.
3. ExpertGroup – экспертная группа, к которой относится пользователь на основании оставленных комментариях о товарах (имя группы и пользователи, относящиеся к этой группе). Например, название «Эксперт по холодильникам», «Специалист по огурцам». Пользователь может состоять во многих группах. В одной группе может быть много пользователей.

Полезные ссылки

- Шутки про SQL
https://laughlore.com/sql-jokes/

- Виды JOIN
https://www.scaler.com/topics/sql/joins-in-sql/

ЗАКЛЮЧЕНИЕ

TEL-RAN
by Starta Institute