***Java DataBase Connectivity (JDBC)*** – это механизм взаимодействия *Java* с БД через универсальный интерфейс (как например *USB*). Он состоит из драйвера, который взаимодействует с конкретной БД, и клиентской частью. Программист взаимодействует с клиентом *JDBC* через пишет *SQL*-запросы. Внутри фреймворка *Spring* применяется реализация ***JdbcTemplate***.

Первый этап – подключение приложения к СУБД через драйвер *JDBC*. Для настройки подключения используется строка *URL* с адресом и параметрами подключения к БД.

Второй этап – **маппинг** (отображение представления). Это преобразование полученных из БД данных в типы Java по определенным правилам. Преобразованные данные уже не связаны с БД и их можно передавать для обработки.

Для работы с *JDBC* и *PostgreSQL* нужно подключить следующие зависимости в *pom.xml*:

*<dependency>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-data-jdbc</artifactId> // пакет-стартер*

*</dependency>*

*<dependency>*

*<groupId>org.postgresql</groupId>*

*<artifactId>postgresql</artifactId>*

*<version>42.3.3</version>*

*</dependency>*



Разберем подробнее *url*-строку подключения к БД:

1. **Схема**. Указывает протокол подключения к ресурсу.
2. **Суб-схема**. Указывает тип БД.
3. **Символы** ***//*** отмечают начало адреса ресурса.
4. **Адрес хоста** **и** отделенный от него двоеточием **порт** сервера БД.
5. **Адрес БД**.
6. В БД H2 между 3 и 4 пунктом можно указать **место хранения** ***file:*** или ***mem:***.

Пакет-стартер позволяет настраивать подключение к БД через файл *application.properties*. Зададим их.

*spring.datasource.url=jdbc:postgresql://localhost:5432/cats*

*spring.datasource.username=kitty*

*spring.datasource.password=purrrrrr*

*spring.datasource.driver-class-name=org.postgresql.Driver*

Для отключения автоконфигурации подключения к БД используется параметр ***exclude***:

*@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)*

При необходимости подключения нескольких БД потребуется использовать альтернативный вариант – вручную создать класс с шаблоном подключения:

*import org.springframework.jdbc.core.JdbcTemplate;*

*import org.springframework.jdbc.datasource.DriverManagerDataSource;*

*import org.springframework.stereotype.Component;*

*@Component*

*public class ManualJdbcConnectService {*

*public static final String JDBC\_URL="jdbc:mysql://cat.world:3306/allcats";*

*public static final String JDBC\_USERNAME="iamacat";*

*public static final String JDBC\_PASSWORD="iamapet";*

*public static final String JDBC\_DRIVER="org.mysql.jdbc.Driver";*

*public JdbcTemplate getTemplate() {*

*DriverManagerDataSource dataSource = new DriverManagerDataSource();*

*dataSource.setDriverClassName(JDBC\_DRIVER);*

*dataSource.setUrl(JDBC\_URL);*

*dataSource.setUsername(JDBC\_USERNAME);*

*dataSource.setPassword(JDBC\_PASSWORD);*

*JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);*

*return jdbcTemplate; } }*

Паттерн ***DAO (Data Access Object)*** предлагает выносить классы по работе с базой данных в отдельный пакет и создавать интерфейсы для них. Такой слой приложения называется ***Data Access Layer (DAL)*** – слой доступа к данным. Для каждого объекта, взаимодействующего с БД, создается свой ***DAO*-класс**, в котором будут выполняться запросы к БД. Таким образом приложение изолируется от исходных данных БД и получает уже преобразованные данные.

Объекты слоя доступа к данным не должны обращаться выше к объектам слоя доступа к сервисам и т.д. Между собой они также желательно не должны быть связаны. Если требуется использовать маппер объекта из другого *DAO*-класса, то можно либо вынести его в отдельный класс, либо сделать статическим. Чем меньше запросов к БД – тем лучше.

Таким образом, у нас есть модель (***class User***), класс-DAO (состоящий из ***interface UserDao*** и его реализации ***class UserDaoImpl***).

*@Data*

*public class User {*

*private String id;*

*private String username;*

*private String nickname; }*

*public interface UserDao {*

*Optional<User> findUserById(String id); }*

*@Component*

*public class UserDaoImpl implements UserDao {*

*private final JdbcTemplate jdbcTemplate;*

*public UserDaoImpl(JdbcTemplate jdbcTemplate){*

*this.jdbcTemplate=jdbcTemplate; }*

*@Override*

*public Optional<User> findUserById(String id) {*

*// реализацию добавим немного позже*

*return Optional.empty(); } }*

Не забываем обновить бины и методы в ***UserService*** и ***UserController***.

*@Service*

*public class UserService {*

*private final UserDao userDao;*

*public UserService(UserDao userDao) {*

*this.userDao = userDao; }*

*public Optional<User> findUserById(String id) {*

*return userDao.findUserById(id); } }*

*@RestController*

*@RequestMapping("/users")*

*public class UserController {*

*private final UserService userService;*

*public UserController(UserService userService) {*

*this.userService = userService; }*

*@GetMapping("/{login}")*

*public Optional<User> getUser(@PathVariable String login){*

*return userService.findUserById(login); } }*

Для получения данных из БД можно использовать метод ***jdbcTemplate.queryForRowSet*** с параметром в виде строки *SQL*-запроса. Он позволяет избежать *SQL*-инъекций и возвращает данные в виде класса-обертки ***SqlRowSet*** с возможностью итерации. Этот класс позволяет обратиться к колонкам строк по имени или по порядковому номеру.

Класс-обертка ***Optional*** позволяет удобно обработать возможные null-значения.

*@Override*

*public Optional<User> findUserById(String id) {*

*SqlRowSet userRows = jdbcTemplate.queryForRowSet("select \* from cat\_user where id = ?", id);*

*if(userRows.next()) {*

*User user = new User( userRows.getString("id"),*

*userRows.getString("username"),*

*userRows.getString("nickname") );*

*return Optional.of(user);*

*} else {*

*log.info("Пользователь с идентификатором {} не найден.", id);*

*return Optional.empty(); } }*

Также существуют методы ***jdbcTemplate.query***, ***jdbcTemplate.queryForList***, ***jdbcTemplate.queryForStream*** и другие.

Для автоматизации преобразования сырых строк из БД в объекты используются реализации интерфейса ***RowMapper***, где *resultSet* – строка с порядковым номером *rowNum*.

*public interface RowMapper<T> {*

*T mapRow(ResultSet resultSet, int rowNum) throws SQLException; }*

Для более сложных случаев у класса ResultSet есть методы по программной обработке любого набора колонок и полей – ***getMetaData()***, ***getColumnCount()***, ***getColumnName()*** и другие.

Рассмотрим применение преобразователя на примере ***RowMapper<Post>***:

*@Override*

*// метод принимает в виде аргумента строку запроса, маппер и параметр SQL-запроса*

*public Collection<Post> findPostsByUser(userId) {*

*String sqlQuery = "select \* from cat\_post where author\_id = ? order by creation\_date desc";*

*return jdbcTemplate.query(sqlQuery, this::makePost, userId); }*

*// собираем объект через маппер*

*private Post makePost(ResultSet resultSet, int rowNum) throws SQLException {*

*return Post.builder()*

*.id(resultSet.getInt("id")*

*.description(resultSet.getString("description"))*

*.photoUrl(resultSet.getString("photo\_url"))*

*.creationDate(resultSet.getDate("creation\_date").toLocalDate())*

*.build();*

Если мы добавляем данные и хотим получить значение автоинкремента, которое присваивается новой записи, требуется использовать класс ***KeyHolder***.

*public long saveAndReturnId(Employee employee) {*

*String sqlQuery = "insert into employees(first\_name, last\_name, yearly\_income) values (?, ?, ?)";*

*KeyHolder keyHolder = new GeneratedKeyHolder();*

*jdbcTemplate.update(connection -> {*

*// Указываем имя столбца, которое хотим получить*

*PreparedStatement stmt = connection.prepareStatement(sqlQuery, new String[]{"id"});*

*stmt.setString(1, employee.getFirstName());*

*stmt.setString(2, employee.getLastName());*

*stmt.setLong(3, employee.getYearlyIncome());*

*return stmt;*

*}, keyHolder);*

*return Objects.requireNonNull(keyHolder.getKey()).longValue(); } // перестраховка*

Подробнее про использование *SQL*-запросов из *java* можно почитать [тут](https://springframework.guru/spring-jdbctemplate-crud-operations/).

Для ускорения выполнения *SQL*-запросов используются транзакции и пакетные вставки (***batch inserts***). Подробности [тут](https://www.baeldung.com/spring-jdbc-batch-inserts) и [тут](https://docs.spring.io/spring-framework/docs/5.3.25-SNAPSHOT/reference/html/data-access.html#jdbc-batch-classic).

*final String sqlQuery = "INSERT INTO FILMS\_GENRES VALUES (?, ?)";  
List<Object[]> batch = new ArrayList<>();  
film.getGenres().stream()  
 .map(Genre::getId)  
 .distinct()  
 .forEach(genreId -> batch.add(new Object[]{ film.getId(), genreId }));  
jdbcTemplate.batchUpdate(sqlQuery, batch);*

Файл ***schema.sql*** содержит все *SQL*-запросы для создания БД, который исполняется каждый раз при старте приложения.

Файл ***data.sql*** содержит все *SQL*-запросы для первичного наполнения БД, который исполняется каждый раз при старте приложения.

Наполнить БД можно и через бины ***CommandLineRunner*** и ***ApplicationRunner*** в классе конфигурации.

*@Config*

*public class StudentConfig {*

*@Bean*

*CommandLineRunner commandLineRunner(StudentRepository repo) {*

*return args -> {*

*repo.save(new Student(“Marian”, 21));*

*repo.save(new Student(“Alex”, 24));*

*}; } }*

***Object-Relational Mapping (ORM)*** – это концепция, которая описывает как связаны понятия из реляционных БД с понятиями из высокоуровневых языков программирования. *ORM* обеспечивает полноценную работу по созданию, изменению и обработке данных из БД. Эти преобразования позволяют сохранить взаимосвязи понятий бизнес-логики и взаимодействовать с ними как с *Java*-объектами.

* **Таблица –> класс**. Таблица – это набор колонок с определенными типами данных. Точно также классы Java содержат поля определенных типов.
* **Колонка таблицы –> поле класса**. Они оба хранят информацию о названии и типе значения.
* **Строка таблицы –> объект (экземпляр класса)**. Строка таблицы содержит конкретные значения, в Java эти данные хранятся в виде объектов.
* **Ячейка –> значение**. Значения из ячеек таблиц записываются в соответствующие по имени и типу поля объекта.
* **Результат –> список объектов**. Запрошенные у БД данные всегда возвращаются в виде таблиц, которые преобразовываются в список объектов из классов и реальных данных.



Ситуация, в которой структура БД не может быть отражена однозначно в виде классов *Java*, называется ***ORM gap***. Например, коллекция юзеров с их постами или коллекция постов со значениями юзеров.

***ORM*-фреймворки** (*Hibernate, MyBatis, Entity*) позволяют автоматизировать процесс получения данных (написание запросов, обработка их параметров, создание мапперов, преобразование результата в объекты и связывание их в сложные структуры). Из минусов – замедление скорости обработки данных.

***JPA* (*Java Persistence API*)** – спецификация, которая содержит требования к современным *ORM*-фреймворкам. Также это кеширование сущностей, управление связями объектов, отслеживание изменений в связанных объектах (для оптимизации), управление транзакциями (гарантия выполнения пакета запросов, или отмена всех).

Для начала работы с ORM нужно добавить три зависимости:

*<!--обеспечивает интеграцию Spring-->*

*<dependency>*

*<groupId>org.springframework.data</groupId>*

*<artifactId>spring-data-jpa</artifactId>*

*<version>2.6.1</version>*

*</dependency>*

*<!--отвечает за реализацию интерфейсов JPA-->*

*<dependency>*

*<groupId>org.hibernate</groupId>*

*<artifactId>hibernate-core</artifactId>*

*<version>5.4.17.Final</version>*

*</dependency>*

*<!--содержит драйвер БД PostgreSQL-->*

*<dependency>*

*<groupId>org.postgresql</groupId>*

*<artifactId>postgresql</artifactId>*

*<version>42.3.2</version>*

*</dependency>*

Методы, помеченные аннотацией ***@Bean***, будут возвращать объекты – бины *Spring*.

Для ручной настройки нужно создать класс *PersistenceConfig* с аннотацией ***@Configuration*** :

* В нем создать бин *DataSource* по аналогии с *JDBC* (все данные берутся из *application.properties*).
* В нем создать метод *hibernateProperties*, возвращающий *Properties*. Свойство *hibernate.dialect* нужна для обеспечения автоматической генерации *SQL*-кода. Свойство *hibernate.show\_sql* отвечает за включение/отключение режима отладочного вывода *SQL*-запроса.
* В нем создать бин-фабрику *\*\*\*EntityManager\*\*\**. Класс *HibernateJpaVendorAdapter* связывает интерфейсы *JPA* и их реализацию внутри фреймворка *Hibernate*.
* В нем создать бин *transactionManager* менеджер транзакций.

В итоге получится следующий класс конфигурации:

*@Configuration*

*@PropertySource(value = "classpath:application.properties")*

*public class PersistenceConfig {*

*private final Environment environment;*

*public PersistenceConfig(Environment environment) {*

*this.environment = environment; }*

*private Properties hibernateProperties() {*

*Properties properties = new Properties();*

*properties.put("hibernate.dialect", environment.getRequiredProperty("hibernate.dialect"));*

*properties.put("hibernate.show\_sql", environment.getRequiredProperty("hibernate.show\_sql"));*

*return properties; }*

*@Bean*

*public DataSource dataSource() {*

*DriverManagerDataSource dataSource = new DriverManagerDataSource();*

*dataSource.setDriverClassName(environment.getRequiredProperty("jdbc.driverClassName"));*

*dataSource.setUrl(environment.getRequiredProperty("jdbc.url"));*

*dataSource.setUsername(environment.getRequiredProperty("jdbc.username"));*

*dataSource.setPassword(environment.getRequiredProperty("jdbc.password"));*

*return dataSource; }*

*@Bean*

*public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {*

*final LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityManagerFactoryBean();*

*emf.setDataSource(dataSource);*

*emf.setPackagesToScan("ru.practicum");*

*final HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();*

*emf.setJpaVendorAdapter(vendorAdapter);*

*emf.setJpaProperties(hibernateProperties());*

*return emf; }*

*@Bean*

*public JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {*

*JpaTransactionManager transactionManager = new JpaTransactionManager();*

*transactionManager.setEntityManagerFactory(entityManagerFactory);*

*return transactionManager; } }*

Аналогичную настройку можно выполнить автоматически с помощью Spring Boot. Подробнее об этом можно почитать [тут](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#dependencies) и [тут](https://www.baeldung.com/spring-boot-starters#JPA). По сути нужно только внедрить зависимости *spring-boot-starter-data-jpa* и *postgresql*, а также в *application.properties* задать параметры *spring.datasource.\** и *hibernate*.

Аннотация ***@Entity*** позволяет пометить простой *java*-класс как сущность для сканирования *ORM*. Сущности характеризуют часть предметной области (пользователь, документ, процесс).

Аннотация ***@Table*** связывает класс с таблицей. Необязательный параметр ***name*** уточняет её имя. Необязательный параметр ***schema*** в *PostgreSQL* по умолчанию использует *public*. Необязательный параметр ***uniqueConstraints*** позволяет указать поля, которые должны быть уникальными при автогенерации таблицы.

Аннотация ***@Id*** связывает поле с одноименной колонкой в таблице. Наличие обязательно, уникальность обеспечивается средствами БД.

Аннотация ***@GeneratedValue*** помечает автоинкрементный идентификатор. В параметре ***strategy*** указывается нужный способ создания новых идентификаторов. О его значениях можно почитать [тут](https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/generatedvalue).

Аннотация ***@Column*** связывает поле с колонкой. Необязательный параметр ***name*** её имя. Необязательный параметр ***nullable = false*** запрещает передавать в БД *null*-значения. Необязательный параметр ***length*** позволяет ограничить длину значения поля БД.

Для корректного сохранения даты метод должен возвращать один из типов – *Instant*, *LocalDate* или *LocalDateTime*. Маппинг происходит на основе возвращаемого типа.

Аннотация ***@Transient*** помечает поля и методы, которые не должны проецироваться на таблицу в БД.

Аннотация ***@Enumerated*** подсказывает фреймворку, каким образом сохранять значение в БД. Первый способ ***EnumType.STRING*** сохраняет в БД строку, полученную в результате вызова метода *enum*-класса *state.name()*. Второй способ ***EnumType.ORDINAL*** сохраняет в БД порядковый номер элемента перечисления, полученную в результате вызова метода *enum*-класса *state.ordinal()*.

*@Entity*

*@Table(name = "users", schema="public", uniqueConstraints=@UniqueConstraint(columnNames = {“id”, “email”})*

*public class User {*

*@Id*

*@GeneratedValue(strategy = GenerationType.IDENTITY)*

*private Long id;*

*@Column(name = "first\_name", nullable = false)*

*private String firstName;*

*@Column(name = "last\_name")*

*private String lastName;*

*@Column(name = "registration\_date")*

*private Instant registrationDate = Instant.now();*

*@Transient*

*private String email;*

*@Enumerated(EnumType.STRING)*

*private UserState state; }*

*public enum UserState {*

*ACTIVE, BLOCKED, DELETED; }*

Использование *EntityManager* и других механизмов автоматизации *ORM* приводит к противоречию между пониманием идентичности объекта в БД и объекта в памяти программы. Например, значение *id == null* у нескольких созданных объектов может запутать менеджера сущностей, поэтому нужно быть аккуратнее с методами *equals()* и *hashCode()*.

Аннотация ***@ElementCollectiont*** указывает на связь таблиц. По умолчанию использует ***LAZY***.

Аннотация ***@CollectionTable*** определяет какую именно таблицу нужно использовать, а также по какой колонке идет связь. Подробнее можно почитать [тут](https://en.wikibooks.org/wiki/Java_Persistence/Relationships).

Свяжем таблицы *Item* и *tags*, как будто *tags* список строк одного объекта *Item*:

*@Entity*

*@Table(name = "items")*

*class Item {*

*@Id*

*@GeneratedValue(strategy = GenerationType.IDENTITY)*

*private Long id;*

*…*

*@ElementCollection*

*@CollectionTable(name="tags", joinColumns=@JoinColumn(name="item\_id"))*

*@Column(name="name") // выберем имя колонки со значением*

*private Set<String> tags = new HashSet<>();*

*@Override*

*public boolean equals(Object o) {*

*if (this == o) return true;*

*if (!(o instanceof Item)) return false;*

*return id != null && id.equals(((Item) o).getId()); }*

*@Override*

*public int hashCode() {*

*return getClass().hashCode(); } }*

Для включения возможности **автогенерирования таблиц** в БД с помощью *Hibernate* нужно добавить следующую строку в *application.properties*:

*spring.jpa.hibernate.ddl-auto=update*

Аннотация ***@OneToOne*** создает связь один-ко-одному для аннотируемого элемента. Это значит, что одному экземпляру класса будут соответствовать один аннотируемый элемент. По умолчанию использует ***EAGER***.

Аннотация ***@OneToMany*** создает связь один-ко-многим для аннотируемого элемента. Это значит, что одному экземпляру класса будут соответствовать коллекция аннотируемых элементов. По умолчанию использует ***LAZY***.

Аннотация ***@ManyToOne*** создает связь многие-к-одному для аннотируемого элемента. Это значит, что одному аннотируемому элементу будут соответствовать множество экземпляров класса. По умолчанию использует ***EAGER***.

Аннотация ***@ManyToMany*** создает связь многие-ко-многим для аннотируемого элемента. По умолчанию использует ***LAZY***.

Необязательный параметр ***cascade*** обозначает, что необходимо выполнять каскадно сразу все операции. Может принимать значения ***ALL***, ***PERSIST***, ***MERGE***, ***REMOVE***, ***REFRESH***, ***DETACH***.

Необязательный параметр ***orphanRemoval*** очень похож на *cascade*, но также удаляет связанную сущность из таблицы БД. Подробнее можно почитать [тут](https://sysout.ru/tipy-cascade-primer-na-hibernate-i-spring-boot/).

Необязательный параметр ***fetch*** обозначает порядок загрузки составных сущностей. Значение ***EAGER*** принудительно загружает все данные, связанные с основной сущностью. Значение ***LAZY*** загружает данные, связанные с основной сущностью, только тогда, когда они понадобятся. Подробнее можно почитать [тут](https://javarush.com/quests/lectures/questhibernate.level14.lecture01).

Для гарантированной выгрузки связанных данных за один запрос, используется выражение ***JOIN FETCH*** при написании *JPQL*-запросов. Выражения ***JOIN LAZY*** не существует.

*public interface PostRepository extends JpaRepository<Post, Long> {*

*@Query( " select p " +*

*"from Post p " +*

*"JOIN FETCH p.author") // JOIN FETCH гарантирует немедленную выборку*

*List<Post> findAllWithAuthors(); }*

Устанавливаемая связь может быть:

* **Однонаправленной** - указывается только на одной сущности, вторая сущность не имеет поля, однозначно связанного с первой.
* **Двунаправленной** - указывается в обоих сущностях.

Аннотация ***@JoinColumn*** связывает текущий экземпляр с дочерними сущностями по колонке *name*. Если названия связываемых колонок отличаются, то используется необязательный параметр ***referencedColumnName***.

*@OneToMany(cascade = CascadeType.All, fetch = FetchType.LAZY, orphanRemoval = true)*

*@JoinColumn(name = “tutorial\_id”)*

*private Set<Comment> comments = new HashSet<>();*

Аннотация ***@OnDelete*** позволяет каскадно удалять дочерние сущности.

*@ManyToOne(cascade = CascadeType.REFRESH)*

*@JoinColumn(name = “from\_accound\_id”)*

*@OnDelete(action = OnDeleteAction.CASCADE)*

*protected Account fromAcc;*

Паттерн ***DTO (Data Transfer Object)*** предлагает использовать специальные объекты, чтобы отделить процесс передачи и получения данных от процесса хранения и загрузки сущностей в БД, а также оптимизировать передачу данных. *DTO* предназначены для использования в контроллерах.

Например, класс *User* хранит логин и пароль. Разработчик создает класс *UserDTO*, в котором нет поля с паролем и других лишних полей. Для преобразования сущности в *DTO* и обратно, вводится специальный **класс-маппер**. Подробнее можно почитать [тут](https://www.baeldung.com/java-dto-pattern).

Внутри *ORM*-фреймворка объектами-сущностями управляет механизм ***EntityManager***. В *Hibernate* его реализация называется ***Session***.



Цикл жизни сущности описан в [спецификации *JPA*](https://jakarta.ee/specifications/persistence/3.0/jakarta-persistence-spec-3.0.html#a1062) и складывается из пяти состояний:

* Статус ***New*** имеют объекты, у которых идентификатор пока что не заполнен. Они еще не были сохранены или загружены из БД. После сохранения и присвоения идентификатора сущность попадает под контроль менеджера.
* Статус ***Managed*** имеют управляемые объекты, вокруг которых выстроен контролирующий слой. Он позволяет отслеживать изменения внутри объекта, связи с другими объектами и т.д.
* Статус ***Detached*** имеют отделенные сущности, о которых менеджер еще помнит, но уже не контролирует их изменения. Например, сущности отправленные на фронтенд можно потом вернуть под управление.
* Статус ***Removed*** имеют сущности, которые будут удалены из БД после фиксации транзакции.
* Статус ***Persisted*** имеют сущности, которые будут сохранены в БД после фиксации транзакции.

Спецификация *JPA* предлагает три способа работы с *SQL*-запросами:

* Использовать нативные (родные) *SQL*-запросы. Нужный результат будет формироваться непосредственно на языке БД.
* Использовать универсальные *SQL*-запросы. Для этого разработчики придумали язык ***JPQL (Java Persistence Query Language)***.
* Использовать инструмент формирования запросов к сущностям ***Criteria API***. Подробнее [тут](https://www.objectdb.com/java/jpa/query/jpql/expression).

Рассмотрим вариант с *Criteria API*. Объект типа *Root<User>* будет содержать обращение к полям сущности. Вызов осуществляется через *select(root)*. Запрос к БД отправляется через *entityManager.createQuery(cr)*.

*@Component*

*public class UserRepositoryImpl implements UserRepository {*

*private EntityManager entityManager;*

*public UserRepositoryImpl(EntityManager entityManager) {*

*this.entityManager = entityManager; }*

*public List<User> searchByEmailDomain(String domain) {*

*CriteriaBuilder builder = entityManager.getCriteriaBuilder();*

*CriteriaQuery<User> query = builder.createQuery(User.class);*

*Root<User> root = query.from(User.class);*

*query.select(root).where(builder.like(root.get("email"), "%"+domain));*

*List<User> foundUsers = entityManager.createQuery(query).getResultList();*

*return foundUsers; }*

*@Override*

*public User save(User user) {*

*entityManager.persist(user); // Метод сохранения*

*return user; } }*

Чтобы увидеть сгенерированные в репозиториях и *ORM*-фреймворках запросы, нужно добавить в *application.properties* параметр *hibernate.show\_sql = true*.

Также будут полезны следующие настройки:

*# вывод запросов*

*logging.level.org.hibernate.SQL=DEBUG*

*# вывод значений, которые подставляются в параметры запросов*

*logging.level.org.hibernate.type.descriptor.sql.BasicBinder = TRACE*

Напомним, что *DAO* отвечает за подключение к БД, запросы данных и их маппинг в сущности. Подключение к БД автоматизированы на уровне фреймворка *Spring*, запросы генерируются через *ORM*, маппинг автоматически выполняется фреймворком с помощью аннотаций *JPA*.

Аннотация ***@EnableJpaRepositories*** в конфигурационном классе приложения включает специальный механизм автоматической имплементации всех *CRUD* методов *DAO*. При старте пакета механизм автоматически проверит каждый класс в этом пакете и подпакетах на наличие репозиториев, унаследованных от интерфейса ***JpaRepository<T, ID>***, сгенерирует их и создаст бины. В первом параметре указывается класс сущности, во втором – тип идентификатора сущности.

*@EnableJpaRepositories(basePackages = "ru.practicum")*

*public class PersistenceConfig { … }*

*public interface UserRepository extends JpaRepository<User, Long> { }*

В результате наследования *UserRepository* получит следующие методы. Подробнее [тут](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.definition).



Среди них есть метод постраничной загрузки данных из таблицы с одноименной сортировкой – ***findAll(Pageable)***. Рассмотрим пример его использования.

*@Service*

*public class UserServiceImpl implements UserService {*

*private final UserRepository repository;*

*public UserServiceImpl(UserRepository repository) {*

*this.repository = repository; }*

*public void checkUsers(){*

*// создаём описание сортировки по полю id*

*Sort sortById = Sort.by(Sort.Direction.ASC, "id");*

*// создаём описание первой "страницы" размером 32 элемента*

*Pageable page = PageRequest.of(0, 32, sortById);*

*do {*

*// запрашиваем у базы данных страницу с данными*

*Page<User> userPage = repository.findAll(page);*

*// результат запроса получаем с помощью метода getContent()*

*userPage.getContent().forEach(user -> { // проверяем пользователей });*

*// проверяем, существует ли следующая страница*

*if(userPage.hasNext()){*

*// если существует, создаём её описание*

*page = PageRequest.of(userPage.getNumber() + 1, userPage.getSize(), userPage.getSort());*

*// или для простоты -- userPage.nextOrLastPageable()*

*} else {*

*page = null; }*

*} while (page != null); } }*

**Транзакция** – это способ выполнения запросов, гарантирующий, что в случае ошибки все сделанные в рамках транзакции изменения будут отменены. Она работает по принципу “все или ничего”. Для обозначения характеристик транзакций используется аббревиатура ***ACID***:

* ***A****tomicity* (атомарность) – все операции транзакции выполняются атомарно, то есть целиком. В случае неудачи одной из операций, транзакция считается неуспешной и полностью откатывается. Если все инструкции выполнены корректно, транзакция считается завершенной, и её результаты фиксируются в БД.
* ***C****onsistency* (согласованность) – транзакция переводит БД из одного согласованного состояния в другое – все правила и ограничения выполняются.
* ***I****solation* (изолированность) – каждая транзакция выполняется сама по себе, без взаимодействия с другими. Ни одна транзакция не работает с изменениями, вносимыми другой, пока та не будет завершена.
* ***D****urability* (надежность) – по завершении транзакции все внесенные ею изменения фиксируются в БД. Даже если по окончании транзакции произойдет сбой, после восстановления эти данные будут в БД.

Рассмотрим примеры транзакций на языке *SQL*.

Команда ***BEGIN*** начинает блок транзакции.

Команда ***COMMIT*** фиксирует текущую транзакцию.

*-- шаг 1: начинаем транзакцию*

*BEGIN;*

*-- шаг 2: списываем средства со счёта Софии*

*UPDATE accounts SET balance = balance - 100.00 WHERE name = 'София';*

*UPDATE bank\_balance SET balance = balance - 100.00 WHERE name = (*

*SELECT bank\_name FROM accounts WHERE name = 'София');*

*-- шаг 3: зачисляем средства на счёт Тимофея*

*UPDATE accounts SET balance = balance + 100.00 WHERE name = 'Тимофей';*

*UPDATE bank\_balance SET balance = balance + 100.00 WHERE name = (*

*SELECT bank\_name FROM accounts WHERE name = 'Тимофей');*

*-- шаг 4: завершаем транзакцию*

*COMMIT;*

Команда ***ROLLBACK*** откатывает текущую транзакцию.

*-- шаг 1: начинаем транзакцию*

*BEGIN;*

*-- шаг 2: списываем средства со счёта Софии*

*UPDATE accounts SET balance = balance - 99.99 WHERE name = 'София';*

*-- шаг 3: упс! Что-то пошло не так: не та сумма. Отменяем транзакцию*

*ROLLBACK;*

По умолчанию при обращении к БД используется режим ***autocommit*** (**автофиксация**). Это значит, что при каждом обращении к БД выполняется отдельная транзакция, что сказывается на производительности БД. Единая транзакция также обеспечивает целостность при внесении связанных изменений.

Рассмотрим пример транзакции в *Java* через *JDBC*.

*public class SimpleTransactionalDAO {*

*public void doInTransaction() {*

*Connection connection = dataSource.getConnection();*

*connection.setTransactionIsolation(Connection.TRANSACTION\_READ\_UNCOMMITTED);*

*try (connection) {*

*connection.setAutoCommit(false);*

*// выполнение SQL-запросов для перевода средств со счёта Софии на счёт Тимофея*

*connection.commit();*

*} catch (SQLException e) {*

*connection.rollback(); } } }*

Рассмотрим пример транзакции в *Java* через *JPA*.

В случае чистого Spring, нужно в классе конфигурации *PersistenceConfig* создать бин *transactionManager* (рассматривали ранее) и добавить аннотацию ***@EnableTransactionManagement***, которая включает управление транзакциями и обработку соответствующих аннотаций.

*@Configuration*

*@EnableTransactionManagement*

*@EnableJpaRepositories(basePackages = "ru.practicum")*

*public class PersistenceConfig { … }*

Аннотацией ***@Transactional*** могут помечать транзакционные методы, классы и контроллеры. С её помощью *Spring* найдет и обернет создаваемый бин скрытым кодом, который обеспечит отлов ошибок и автоматическое завершение или отмену транзакций. Также можно настроить типы отлавливаемых ошибок транзакции. Подробнее можно почитать [тут](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html).

*@Transactional(readOnly = true, isolation = Isolation.DEFAULT)*

*@Override*

*public List<UserDto> getAllUsers() {*

*return repository.findAll()*

*.stream()*

*.map(UserDto::from)*

*.collect(Collectors.toList()); }*

Необязательный параметр ***readOnly*** задает один из нескольких режимов доступа к БД:

* ***READ WRITE*** (чтение и запись) – значение по умолчанию. Возможно выполнение всех команд.
* ***READ ONLY*** (только чтение) – запрещено выполнение команд *INSERT*, *UPDATE*, *DELETE*, *CREATE*.

Необязательный параметр ***isolation*** задаёт **уровень изоляции** (*isolation level*), который определяет какие данные транзакция может увидеть в процессе выполнения параллельной транзакции:

* ***DEFAULT*** () – уровень по умолчанию. Транзакция создается с уровнем, заданным в настройках БД.
* ***READ\_COMMITTED*** (чтение фиксированных данных) – будут видны только те строки, которые были зафиксированы до начала её выполнения. Предотвращают проблему “грязного” чтения.
* ***REPEATABLE\_READ*** (повторяющееся чтение) – транзакция не увидит изменений данных, которые были внесены после её начала. Предотвращают проблему “грязного” и неповторяющегося чтения.
* ***SERIALIZABLE*** (сохраняемые) – высший уровень изоляции. Транзакции будут выполняться так, как будто параллельных транзакций не существует (последовательно). Из-за этого падает производительность.
* ***READ\_UNCOMMITTED*** (чтение нефиксированных данных) – низший уровень изоляции. При изменении одной и той же строки таблицы несколькими транзакциями, в окончательном варианте строка будет иметь значение, которое определено всем набором успешно выполненных транзакций. В *PostgreSQL* этот уровень не реализован. На нем возможны все возможные проблемы.

Установить режим доступа и уровень изоляции также можно с помощью [*SQL*-команды](https://postgrespro.ru/docs/postgrespro/12/sql-set-transaction)

*SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ WRITE;*

Проблемы выполнения параллельных транзакций:

* **“Грязное” чтение** (*dirty reads*) – в результатах запроса появляются промежуточные результаты параллельной транзакции, которая еще не завершилась.
* **Фантомное чтение** (*phantom reads*) – в результатах повторяющегося запроса появляются и исчезают строки, которые в данный момент модифицирует параллельная транзакция.
* **Неповторяющееся чтение** (*non-repeatable reads*) – запрос с одними и теми же условиями дает неодинаковые результаты в рамках транзакции.
* **Потерянное обновление** (*lost update*) – две параллельные транзакции меняют одни и те же данные, при этом итоговый результат обновления предсказать невозможно.

В случае, когда транзакционный метод контроллера *UserController* вызывает транзакционный метод сервиса *UserServiceImpl* внутри себя, такая транзакция называется **вложенной**. БД не знакома с понятием вложенной транзакции, поэтому все проблемы должны решаться на уровне кода приложения. Для работы со вложенными транзакциями применяется необязательный параметр ***propagation*** (распространение) с одним из следующих значений:

* ***REQUIRED*** (необходимая) – выполнять метод в текущей транзакции. Если транзакции не существует, *Spring* создаст новую. Используется по умолчанию.

*nextTransaction = thisTransaction.exists() ? thisTransaction : new Transaction();*

* ***SUPPORTS*** (поддерживается) – выполнять метод в режиме транзакции нужно только тогда, когда она уже существует. Новая транзакция не создается.

*nextTransaction = thisTransaction.exists() ? thisTransaction : null;*

* ***MANDATORY*** (обязательная) – выполнять метод в текущей транзакции. Если транзакции не существует, метод сгенерирует исключение.

*if(!thisTransaction.exists()) throw new Exception();*

* ***NEVER*** (никогда) – не выполнять метод в существующей транзакции. Полностью исключает использование метода в транзакциях.

*if(thisTransaction.exists()) throw new Exception();*

* ***NOT\_SUPPORTED*** (не поддерживается) – существующая транзакция будет приостановлена и метод выполнится без транзакции.

*thisTransaction.suspend(); nextTransaction = null;*

* ***REQUIRES\_NEW*** (требуется новая) – существующая транзакция будет приостановлена и создана новая. Метод будет выполнен в рамках новой транзакции.

*thisTransaction.suspend(); nextTransaction = new Transaction();*

* ***NESTED*** (вложенная) – существующая транзакция будет приостановлена и создаст в ней **точку сохранения**, если вызываемый метод сгенерирует исключение.

*thisTransaction.savepoint(); nextTransaction = new Transaction();*

Эти режимы позволили разделить транзакции на **логические** (выполняемые и вызываемые транзакционные методы классов и бинов *Spring*) и **физические** (выполняемые *SQL*-запросы внутри транзакции БД).

*Spring Data* позволяет генерировать методы для работы с репозиторием с помощью технологии **запросных методов** (*Query Methods*). Достаточно написать название метода в виде понятной команды и *Spring Data* преобразует его в запрос. Есть возможность составлять логические выражения с помощью *And* и *Or*, а также *LessThan*, *Between*, *Containing* и других. Результат может быть в виде списка или единичной сущности (*Optional* поддерживается). Из минусов – не предусмотрена расстановка скобок и использование агрегирующих операторов. Подробнее можно почитать [тут](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation).

*interface UserRepository extends JpaRepository<User, Long> {*

*// найти юзеров, у которых поле email содержит строку-параметр и игнорировать регистр*

*List<User> findByEmailContainingIgnoreCase(String emailSearch);*

*// найти юзеров, у которых поле email содержит строку-параметр и игнорировать регистр*

*// и вывести только firstName и email*

*List<UserShort> findAllByEmailContainingIgnoreCase(String emailSearch);*

*// найти юзеров со статусом states, зарегистрированных в промежутке from и to,*

*// и отсортировать по возрастанию*

*List<User> findAllByStateInAndRegistrationDateBetweenOrderByIdAsc(Set<UserState> states, Instant from, Instant to); }*

Для получения ограниченной информации из репозитория через запросные методы нужно задействовать **механизм проекций**. Он предполагает создание интерфейса-проекции, соответствующей той сущности, поля которой нужно получить в результате запроса. Доступ к этим полям обеспечивается через методы-геттеры. Подробнее можно почитать [тут](https://www.baeldung.com/spring-data-jpa-projections).

*public interface UserShort {*

*String getFirstName();*

*String getEmail(); }*

*interface UserRepository extends JpaRepository<User, Long> {*

*// найти юзеров, у которых поле email содержит строку-параметр и игнорировать регистр*

*// и вывести только firstName и email*

*List<UserShort> findAllByEmailContainingIgnoreCase(String emailSearch); }*

Рассмотрим пример **создания кастомного метода** для репозитория. Допустим, что в БД хранятся только пользователи, а мы хотим вывести их вместе с ip-адресами почтовых серверов (за один запрос к репозиторию).

* Создадим финальный класс с нужными нам полями.

*@Getter*

*@Setter*

*public class UserShortWithIP implements UserShort {*

*private String firstName;*

*private String email;*

*private String ip;*

*public UserShortWithIP(UserShort user, String ip) {*

*this.name = user.getFirstName();*

*this.email = user.getEmail();*

*this.ip = ip; } }*

* Создадим кастомный интерфейс и укажем его вторым наследуемым интерфейсом репозитория.

*public interface UserRepositoryCustom {*

*List<UserShortWithIP> findAllByEmailContainingIgnoreCaseWithIP(String emailSearch); }*

*interface UserRepository extends JpaRepository<User, Long>, UserRepositoryCustom {*

*List<UserShort> findAllByEmailContainingIgnoreCase(String emailSearch); }*

* Создадим имплементацию кастомного интерфейса с использованием специального слова ***Impl***, через которое фреймворк узнает о пользовательской имплементации. Аннотация ***@Lazy*** указывает *Spring*, что бин внедряет сам себя.

*public class UserRepositoryImpl implements UserRepositoryCustom {*

*private final UserRepository userRepository;*

*public UserRepositoryImpl(@Lazy UserRepository userRepository){*

*this.userRepository = userRepository; }*

*@Override*

*public List<UserShortWithIP> findAllByEmailContainingIgnoreCaseWithIP(String emailSearch) {*

*return userRepository.findAllByEmailContainingIgnoreCase(emailSearch).stream()*

*.map(userShort -> new UserShortWithIP(userShort, getServerIP(getEmailServer(userShort.getEmail()))))*

*.collect(Collectors.toList()); }*

*private String getServerIP(String emailServer) {*

*try {*

*return InetAddress.getByName(emailServer).toString();*

*} catch (UnknownHostException e) {*

*return "127.0.0.1"; } }*

*private String getEmailServer(String email) {*

*String[] emailParts = email.split("\\@");*

*if (emailParts.length != 2 || emailParts[1].isEmpty()) {*

*throw new IllegalArgumentException("неверный адрес почтового сервера"); }*

*return emailParts[1]; } }*

Если имя запросного метода заходит за 30 символов или содержит более 2 параметров, то лучше использовать аннотацию ***@Query***. Она позволяет обойти ограничения запросного метода и использует язык ***JPQL*** (***Java Persistence Query Language***), который преобразуется в запросы на диалекте используемой БД. Особенности диалекта игнорируются. Параметр метода передается через вопросительный знак и его порядковый номер.

*public class ItemCountByUser {*

*private Long userId;*

*private Long count; }*

*interface UserRepository extends JpaRepository<User, Long> {*

*@Query("select new ru.practicum.item.ItemCountByUser(it.userId, count(it.id))" +*

*"from Item as it "+*

*"where it.url like ?1 "+*

*"group by it.userId "+*

*"order by count(it.id) desc")*

*List<ItemCountByUser> countItemsByUser(String urlPart); }*

Аннотация ***@Query*** также используется для написания запроса на языке используемой БД - **нативного запроса**. Она позволяет задействовать особенности конкретного диалекта БД. Для этого используется необязательный параметр ***nativeQuery = true***.

*public class ItemCountByUser {*

*private Long userId;*

*private Long count; }*

*interface UserRepository extends JpaRepository<User, Long> {*

*@Query(value = "select it.user\_id, count(it.id) as count "+*

*"from items as it left join users as us on it.user\_id = us.id "+*

*"where (cast(us.registration\_date as date)) between ?1 and ?2 "+*

*"group by it.user\_id", nativeQuery = true)*

*List<ItemCountByUser> countByUserRegistered(LocalDate dateFrom, LocalDate dateTo); }*

Если задача предполагает необходимость работы с сущностями и сложные условия поиска, но не требует агрегаций, можно использовать язык ***QueryDSL*** (***Query Domain Specific Language***). Для этого нужно добавить зависимость и плагин. После этого *QueryDSL* будет генерировать вспомогательный *Q*-классы. Подробнее можно почитать [тут](http://querydsl.com/static/querydsl/5.0.0/reference/html_single/).

*<dependencies>*

*<dependency>*

*<groupId>com.querydsl</groupId>*

*<artifactId>querydsl-jpa</artifactId>*

*<version>5.0.0</version>*

*</dependency>*

*</dependencies>*

*<build>*

*<plugins>*

*<plugin>*

*<groupId>com.mysema.maven</groupId>*

*<artifactId>apt-maven-plugin</artifactId>*

*<version>1.1.3</version>*

*<executions>*

*<execution>*

*<goals>*

*<goal>process</goal>*

*</goals>*

*<configuration>*

*<outputDirectory>target/generated-sources/java</outputDirectory>*

*<processor>com.querydsl.apt.jpa.JPAAnnotationProcessor</processor>*

*</configuration>*

*</execution>*

*</executions>*

*<dependencies>*

*<dependency>*

*<groupId>com.querydsl</groupId>*

*<artifactId>querydsl-apt</artifactId>*

*<version>5.0.0</version>*

*</dependency>*

*</dependencies>*

*</plugin>*

*</plugins>*

*</build>*

Связка репозиторий + *QueryDSL* заменяет собой связку *EntityManager* + *Criteria API*.

* Первая связка

*class ItemServiceImpl implements ItemService {*

*@Override*

*@Transactional(readOnly = true)*

*public List<ItemDto> getItems(long userId, Set<String> tags) {*

*BooleanExpression byUserId = QItem.item.userId.eq(userId); // предикат*

*BooleanExpression byAnyTag = QItem.item.tags.any().in(tags); // предикат*

*Iterable<Item> foundItems = repository.findAll(byUserId.and(byAnyTag));*

*return ItemMapper.mapToItemDto(foundItems); } }*

* Вторая свзяка

*QItem.item.userId.eq(req.getUserId()).and(QItem.item.tags.any().in(req.getTags()))*

Для автоматизации построения *API* репозиториев создан инструмент ***Spring Data REST***.

* Для его использования на чистом *Spring* нужно подключить зависимость-стартер.

*<dependency>*

*<groupId>org.springframework.data</groupId>*

*<artifactId>spring-data-rest-webmvc</artifactId>*

*<version>3.7.0</version>*

*</dependency>*

*<dependency>*

*<groupId>com.jayway.jsonpath</groupId>*

*<artifactId>json-path</artifactId>*

*<version>2.6.0</version>*

*</dependency>*

* В проекте на чистом *Spring* требуется создать класс конфигурации. Также добавим префикс *api* ко всем эндпоинтам.

*@Configuration*

*@Import({RepositoryRestMvcConfiguration.class})*

*public class RestConfig {*

*@Bean*

*public RepositoryRestConfigurer repositoryRestConfigurer() {*

*return new RepositoryRestConfigurer() {*

*@Override*

*public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config, CorsRegistry cors) {*

*config.setBasePath("/api"); } }; } }*

* Для использования на Spring Boot нужен только стартер ***spring-boot-starter-data-rest***. Префикс *api* добавляется через файл *application.properties*:

*spring.data.rest.basePath=/api*

Теперь, если отправить запрос на эндпоинт ***/api*** приложения, можно увидеть все доступные репозитории и дополнительные служебные ресурсы.

Аннотация ***@RepositoryRestResourse*** включает репозиторий в список *API*. После этого он появится в ответе запроса на эгдпоинт ***/api*** приложения. Необязательный параметр ***path*** позволяет вручную задать URL для репозитория. Подробнее можно почитать [тут](https://docs.spring.io/spring-data/rest/docs/current/reference/html/#customizing-sdr.configuring-the-rest-url-path).

*@RepositoryRestResource*

*interface UserRepository extends JpaRepository<User, Long> { … }*

При обращении к конкретному репозиторию (например, ***/api/users***) мы получим список пользователей, информацию о пагинации, ссылки на конкретные сущности, ссылки для перехода по страницам, ссылки на схему данных и т.д. Все это помогает различным клиентам вашего *API* при обработке данных. Ссылки с префиксом ***/api/progile/*** позволяют получить схему возможных запросов, т.е. описание методов *REST* для указанного репозитория.

Эндпоинты *API* существуют отдельно от ваших собственных эндпоинтов и могут работать вместе без каких-либо проблем. Они уже поддерживают *REST*-методы, а также запросные методы. Получить список запросных методов можно по специальному URL ***/api/<имя\_репозитория>/search***. К сожалению, в этом списке будут отсутствовать кастомные методы.

Аннотация ***@RestResourse*** позволяет изменять имена методов репозитория.

Аннотация ***@Param*** позволяет настроить названия параметров *URL*-запроса.

*@RepositoryRestResource(path = "people")*

*interface UserRepository extends JpaRepository<User, Long> {*

*@RestResource(path = "emails")*

*List<User> findByEmailContainingIgnoreCase(@Param("email") String emailSearch); }*