**Система сборки** – это инструмент, преобразующий исходный код в программу, которую можно запускать. Существует несколько систем сборок:

* ***Apache Ant*** – использует императивный подход с составлением алгоритма действий в *XML*-файле.
* ***Apache Maven*** – использует декларативный подход с описанием конечного результата.
* ***Gradle*** – комбинирует императивный (сценарии сборки) и декларативный (дерево зависимостей) подходы.

Преимущества *Maven*:

* Декларативное описание проекта. Для этого используется файл *pom.xml*, где объявляется, что должно получиться в результате (не требует конкретных команд).
* Автоматическое управление зависимостями. Система сама скачивает зависимости и разрешает конфликты версий.
* Фиксированные циклы сборки проекта, состоящие из определенной последовательности фаз, с достижением конкретных целей в каждой.
* Четкая структура каталогов, где располагаются исходный код, тесты и другие ресурсы.
* Модульная структура, позволяющая легко расширить функциональность проекта.

**Архетип** – это шаблон, который можно использовать в качестве основы для своего проекта. Он помогает стандартизировать разработку и получить доступ к уже готовым решениям.

**Артефакт** – это то, что создано в процессе разработки программного обеспечения. Им может быть любая библиотека, зависимость или плагин.

**Артефакт *Maven*** – это файл *JAR* формата, который появляется при сборке проекта. У него есть своя группа настроек:

* ***groupId*** – уникальный идентификатор группы, которые создали проект. Как правило он основан на полном доменном имени организации (например, *ru.yandex.practicum*).
* ***artifactId*** – уникальное имя артефакта, генерируемого этим проектом.
* ***version*** – версия артефакта.

Ссылка на гайд по основам *XML* - <https://doka.guide/tools/xml/>

Для оформления *pom.xml* также могут использоваться следующие **теги**:

* *<build> </build>* - блок настройки процесса сборки
* *<plugins> </plugins>* - блок настройки плагинов
* *<resources> </ resources>* - блок настройки доп. ресурсов
* *<dependencies> </ dependencies>* - блок настройки зависимостей
* *<dependencyManagement> </ dependencyManagement>* - блок настройки зависимостей в *parent-POM*
* *<goals> </ goals>* - блок настройки задач/целей
* *<profiles> </ profiles>* - блок настройки профилей создания
* *<developers> </ developers>* - блок настройки разработчиков
* *<mailingLists> </ mailingLists>* - блок настройки списков рассылки
* *<parent> </ parent>* - блок настройки родительского проекта
* *<scope> </scope>* - модификатор, указывает на каком этапе сборки используется данный артефакт

Зависимости нужны для того, чтобы работали аннотации. Плагины нужны для изменения процесса компиляции/сборки в *Lifecycle*.

Ссылка на документацию Maven - <https://maven.apache.org/pom.html#Maven_Coordinates>

**Репозиторий *Maven*** – это директория, где хранятся все файлы проекта, библиотеки, плагины и любые другие артефакты. Существуют локальный репозиторий (папка с расширением *.m2*), удаленные репозитории сообществ (через протоколы *https://*, *file://* и другие) и центральный репозиторий Maven Central (<https://repo.maven.apache.org/maven2/>). Инструкции по подключению можно найти на <https://search.maven.org/> и <https://mvnrepository.com/>. Пример подключения:

*<repositories>*

*<repository>*

*<id>additional-local-dir</id>*

*<url>file://path/to/directory</url>*

*</repository>*

*<repository>*

*<id>org-remote-repository</id>*

*<url>https://maindomain.org/maven2</url>*

*</repository>*

*<repositories>*

Приоритеты поиска зависимостей в репозиториях локальный -> центральный -> удаленный.

Процесс сборки включает несколько жизненных циклов. Всего их три:

1. ***clean*** – в ходе этого цикла происходит очистка проекта.
2. ***default*** – основной цикл сборки. В ходе него происходит компиляция кода, генерация тестов, сборка дополнительных артефактов, различного рода проверки и т.д. Включает 23 фазы.
3. ***site*** – посвящен созданию проектной документации.

Каждый цикл включает набор фаз. Есть подготовительные (*pre-*), основные и завершающие (*post-*) фазы. Ссылка на фазы циклов - <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>.

Следующий плагин с помощью команды ***surefire-report:report***создает HTML-страницу с отчетом о пройденных тестах в удобном для чтения виде.

*<plugin>*

*<groupId>org.apache.maven.plugins</groupId>*

*<artifactId>maven-surefire-report-plugin</artifactId>*

*<version>2.22.2</version>*

*</plugin>*

**Дерево зависимостей** (***dependency tree***) – совокупность зависимостей всех компонентов, используемых в проекте. Посмотреть все зависимости можно с помощью команды ***dependency:tree***. Зависимости, которые появились в проекте неявно, называются **транзитивными** (***transitive***) или переходными.

Когда для работы пакетов требуются разные версии одной и той же библиотеки, проект попадает в **ад зависимостей** (***dependency hell***). Из-за этого проект может не запуститься. В качестве решения предлагается файл ***BOM*** (***Bill of Material***) или **список материалов**.

В *maven* реализована возможность **наследования** от *parent*-проекта с помощью тега *<parent></parent>*. Для создания проекта-предка используется строка *<packaging>pom</packaging>*. Изменения в проекте-наследнике имеют приоритет над проектом-предком. Благодаря наследования можно переиспользовать все связанные зависимости. Файл *BOM* является предком для *POM*-файла. Например, с помощью наследования *BOM*-файла можно указать все версии зависимостей и не указывать их (или переопределить версию нужной зависимости в блоке *<properties>*) в *POM*-файле.

При наследовании может быть только один проект-предок и нельзя подключить второй *BOM*-файл. Обойти это ограничение можно, подключив *BOM* как зависимость с использованием строк *<type>pom</type>* и *<scope>import</scope>*.

<https://docs.oracle.com/middleware/1212/core/MAVEN/maven_version.htm#MAVEN8855>

<https://medium.com/javarevisited/10-important-maven-plugins-for-java-developers-330b98b71720>

<https://habr.com/ru/post/205118/>

<https://github.com/eirslett/frontend-maven-plugin>

<https://www.baeldung.com/maven-polyglot>

Гайды на postman <https://habr.com/ru/company/maxilect/blog/596789/>

<https://learning.postman.com/docs/getting-started/introduction/>

<https://spring.io/>

<https://start.spring.io/>

***Spring Boot*** – упрощенная версия ***Sping Framework*** с максимально упрощенной настройкой запуска.

Аннотация ***@SpringBootApplication*** показывает фреймворку *Spring Boot*, что это основной класс с конфигурацией приложения.

Файл ***application.properties*** хранит настройки проекта (стандартные настройки *Spring Boot*, пути к сторонним приложениям и т.д.). Например, *server.port=8081*.

<https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html>

В результате разработки подхода ***API First*** (когда разработка строится вокруг *API*) появился архитектурный стиль ***REST*** (*Representation State Transfer*) или **передача репрезентативного состояния**. Он состоит из ключевых принципов проектирования распределенных систем, заложенных Филдингом:

1. Клиент-серверная архитектура с разделением ответственности (***Client-Server***).
2. Сервер не хранит информацию о состоянии клиента (***Stateless***). Каждый запрос независим, как будто сделан в первый раз.
3. Единообразие интерфейса обращения к серверу с любого типа устройства (***Uniform Interface***).
4. Многоуровневость архитектуры (***Layered system***). Все внутренние взаимодействия также работают по принципу клиент-сервер.
5. Кешируемость (***Cache***). При идентичном запросе данные берутся из кеша, а не повторно запрашиваются у сервера.
6. Код по запросу (***Code on demand***). Функциональность клиента может быть расширена кодом, приходящим с сервера (сценарии оживления страниц).

**Преимущества использования *REST*:**

1. Производительность
2. Масштабируемость
3. Гибкость к изменениям
4. Отказоустойчивость
5. Простота поддержки

Ключевое понятие в *REST* – это **ресурс**, т.е. данные любого типа и формата. Для указания местоположения ресурса используется ***URL***.

**Основные правила именования ресурсов:**

* Использовать существительные
* Слеш указывает на иерархию
* Слеш в конце *URL* лучше не использовать
* Для разделения слов использовать дефисы (пробелы, слитное написание и нижние подчеркивания под запретом)
* Не включать в название ресурса имя *HTTP*-метода

Правила именования ресурсов в *REST* <https://restfulapi.net/resource-naming/>

В большинстве случаев в качестве протокола реализации *REST* используется *HTTP* и его методы:

* ***GET*** получает ресурсы
* ***POST*** создает новый ресурс
* ***PUT*** заменяет существующие данные или, при их отсутствии, создает новый ресурс
* ***PATCH*** используется для частичного обновления данных ресурса
* ***DELETE*** удаляет ресурс
* ***HEAD*** используется для получения заголовков ответа
* ***OPTIONS*** используется для получения списка *HTTP*-методов, которые поддерживает сервер

Метод считается **небезопасным**, если может изменить ресурс. Метод считается **идемпотентным**, если всегда даст один и тот же результат независимо от текущего состояния. Метод *POST* не идемпотентен, т.к. каждый раз создает новый ресурс.

Приложение на базе *Spring Framework* настраивается через **аннотации**. При запуске фреймворк проверяет их наличие и связывает помеченный ими код с определенной функциональностью. Это называется **конфигурацией, основанной на аннотациях**.

**Контроллер** – это специальный класс в приложении, который предназначен для обработки *HTTP*-запросов от клиента и возвращения результатов (т.е. класс для создания эндпоинтов).

Аннотация ***@RestController*** обозначает классы-контроллеры с методами обработки запросов к эндпоинтам *API*. Также автоматически добавляет их в контекст приложения и аннотацию ***@ResponseBody***.

Аннотация ***@RequestMapping*** дает фреймворку понять, по какому пути можно будет отправить запрос к этому методу контроллера. Пути запроса строятся относительно базового пути приложения. Позволяет аннотировать весь класс – в этом случае путь строится путем сложения общей аннотации класса и аннотации конкретного метода.

Начиная со Spring Framework 4.3, на основе аннотации *@RequestMapping* появились новые аннотации ***@GetMapping***, ***@PostMapping***, ***@PutMapping***, ***@DeleteMapping*** и ***@PatchMapping***. Они позволяют указать конкретный метод запроса и отказаться от свойства *method*.

*@RequestMapping(value = "/home", method = RequestMethod.GET) // можно так*

*@GetMapping("/home") // но так проще и удобнее*

Аннотация ***@RequesrBody*** означает, что значение аргумента нужно взять из тела запроса. При этом объект, который пришел в теле запроса, будет автоматически сконвертирован в *Java*-объект. Имеет смысл только в классах-контроллерах.

*public Post create(@RequestBody Post post) {*

*posts.add(post);*

*return post; }*

Аннотация ***@FieldDefaults(level = AccessLevel.Private)*** устанавливает все значения полей в *private*.

Аннотация ***@Valid*** в методе контроллера заставляет его провести валидацию передаваемого объекта согласно его модели.

Аннотация ***@Validated*** в методе контроллера заставляет его провести валидацию передаваемого объекта согласно его модели, поля которого добавлены в ***groups***. Также нужно создать пустой интерфейс-маркер ***Create***. Подробнее можно почитать [тут](https://www.baeldung.com/spring-valid-vs-validated). Альтернативным вариантом будет создание дополнительных *Dto*-классов с различными комбинациями аннотаций валидации.

*public UserDto createUser(@Validated(Create.class) @RequestBody UserDto userDto) {*

*return userService.createUser(userDto);}*

*public class UserDto {*

*@Email(groups = {Create.classs, Update.class})*

*String email; }*

**Лог** (***log***) или **журнал регистрации** – это набор записей о происходящих в программе событиях (ошибках, действиях пользователя и т.д.). Процесс добавления в журнал новых записей называется **логированием**. У каждой записи есть точная метка даты и времени. Из записей лога можно:

* Получить информацию о характеристиках окружения, в котором запущена программа.
* Разобраться с ошибками.
* Устранить проблемы с производительностью.
* Обнаружить подозрительное поведение пользователя и предотвратить злоумышленные действия.
* Предотвратить проблемы с инфраструктурой.

Большинство известных фреймворков логирования в *Java* состоят из трех частей:

1. ***Logger*** (**журнал**) – специальный класс, который преобразует событие в объект и передает его дальше другим компонентам фреймворка.
2. ***Appender*** (**добавлять**) – интерфейс, который отвечает за добавление объекта, полученного от логера, непосредственно в лог. В зависимости от местоположения журнала используются разные реализации. Запись всегда добавляется в конец лога.
3. ***Layout*** (**формат**) – интерфейс, который перед выводом записи из лога в консоль или на печать приводит ее к определенному формату.

В качестве интерфейса единого *API* логирования мы будем использовать фреймворк ***The Simple Logging Façade for Java*** ([***SLF4J***](https://www.slf4j.org/)), а в качестве реализации *API* – фреймворк [***Logback***](https://logback.qos.ch/).

1. Для получения нового объекта класса ***Logger*** используется класс-фабрика ***LoggerFactory*** и его статический метод ***getLogger(String)***. Во время работы программы нет смысла менять экземпляр логера, поэтому можно пометить переменную как final. В отличие от констант, переменную логера принято записывать в нижнемВерблюжьемРегистре.

*private static final Logger log = LoggerFactory.getLogger("Practicum");*

1. В составе *Logback* есть большой набор готовых реализаций типа *Appender*:

* ***ConsoleAppender*** – выводит записи в консоль.
* ***FileAppender*** – сохраняет записи в файл.
* ***RollerFileAppender*** – сохраняет записи в файл, но при этом умеет автоматически создавать новый файл при достижении заданного лимита размера. Лимит задается программистом.

1. В большинстве случаев в *Logback* для интерфейса *Layout* используется реализация по умолчанию ***PatternLayout***. Например, шаблон %d{HH:mm:ss}: %message%n выведет лог в таком виде:

*14:06:49: Событие 1*

*14:27:53: Событие 2 // и так далее*

В *IDEA* по умолчанию вывод лога проекта перенаправляется на консоль с помощью компонента *ConsoleAppender*. Также есть режим ***soft wraps*** — перенос текста на новую строку, если она не помещается в видимую область окна консоли.

Для удобства работы с большими объемами записей в *SLF4J* были созданы пять **специальных уровней важности** и соответствующие им методы ***trace***, ***debug***, ***info***, ***warn***, ***error***:

* ***TRACE*** (**отслеживать**) – для трассировочных сообщений, которые предоставляют очень подробную информацию о каком-либо процессе. На этом уровне может записываться все что происходит в системе (выбор ветки условий, значений параметров внутри циклов, каждый шаг алгоритма и т.д.).
* ***DEBUG*** (**отлаживать**) – для отладочных сообщений, которые важны во время анализа некорректного поведения программы. Чаще всего в таких записях содержится информация о значениях переменных, которые относятся к бизнес-логике приложения.
* ***INFO*** (**информация**) – для сообщений, которые помогают понять текущее состояние программы. Например, информация о статусе обработки какого-либо запроса, результате авторизации пользователя, сведения об *IP*-адресе или сетевом порте, на котором запустилось веб-приложение.
* ***WARN*** (**предупреждать**) – для сообщений с предупреждениями, которые появляются в случае, если что-то пошло не так. Например, было сгенерировано исключение или обнаружена проблема в ходе выполнения алгоритма (допустим переданы данные в неверном формате). Программа может обработать эту ситуацию самостоятельно и продолжить работу.
* ***ERROR*** (**ошибка**) – для сообщений о критических ошибках, требующих оперативного вмешательства, без которого программа не сможет продолжить работу. Например, нехватка ресурсов, невозможность установить соединение с базой данных и т.д.

В процессе логирования можно использовать разные комбинации входных параметров:

* ***String msg*** – логируется только переданное сообщение
* ***String format, Object… arguments*** – логируется переданное сообщение и набор аргументов.

*log.info("Пользователь {} сменил статус на {}.", "Григорий Логинов", "Занят");*

* ***String msg, Throwable t*** – логируется переданное сообщение и полученное исключение. Полезно в блоке *try-catch*.

По умолчанию логируются все сообщения, кроме трассировочных. Однако это можно изменить, указав уровень важности, начиная с которого сообщения будут логироваться. Также можно выключить логирование с помощью уровня ***OFF***. Иерархия уровней выглядит так (от наименее важного к наиболее важному):

*TRACE < DEBUG < INFO < WARN < ERROR*

Для изменения уровня логирования экземпляр класса *Logger* нужно привести к типу логера конкретной системы (в *Lomback* это класс ***ch.qos.logback.classic.Logger***), в котором есть метод ***setLevel(Level newLevel)***. Это можно также сделать в конфигурационном файле *application.properties* через добавление строки:

*logging.level.полное\_имя\_логгера=уровень\_важности*

*logging.level.ru.yandex.practicum=debug // при указании корневого пакета, изменятся уровни всех логеров*

**Стартер** (***starter***) – это набор библиотек и конфигураций, которые специально создается для транзитивного внесения в проект нужных зависимостей. Например, в стартере *spring-boot-starter-web* транзитивно подключается фреймворк логирования *spring-boot-starter-logging*.

Любой стартер от команды *Spring Framework* транзитивно добавляет в проект фреймворки логирования ***logback-core*** и ***slf4j-api***, а в Spring Boot есть специальный стартер с ними *spring-boot-starter-logging*.

Добавить информацию для логирования можно несколькими способами:

1. **Вручную**. Из минусов – много информации придется постоянно обновлять в ручном режиме.

log.info(“Информация для логирования с уровнем INFO.”);

1. **Средствами *Spring Framework***. Требуется изменить уровень логера сервера запросов в конфигурационном файле *application.properties*:

*logging.level.org.springframework.web.servlet.DispatcherServlet=debug*

1. **Через стороннюю библиотеку**. Например, подключить зависимость *LogBook* (логирует в *JSON*), и изменить уровень логирования запросов этим пакетом в конфигурационном файле *application.properties*:

*<dependency>*

*<groupId>org.zalando</groupId>*

*<artifactId>logbook-spring-boot-starter</artifactId>*

*<version>2.14.0</version>*

*</dependency>*

*logging.level.org.zalando.logbook=TRACE*

**Библиотека** [***Lombok***](https://projectlombok.org/) позволяет автоматически генерировать геттеры, сеттеры, конструкторы, методы *toString()*, *equals()* и *hashCode()*, а также автоматизировать создание переменных для логирования с помощью аннотации и многое другое. Подключить зависимость можно так:

*<dependency>*

*<groupId>org.projectlombok</groupId>*

*<artifactId>lombok</artifactId>*

*<version>1.18.20</version>*

*<scope>provided</scope>*

*</dependency>*

*Lombok* используется только на этапе компиляции программы. На это указывает тег ***provided***.

[Полный список аннотаций *Lombok*](https://projectlombok.org/features/all)

Аннотации ***@Getter*** и ***@Setter*** перед переменной/классом позволяют автоматически генерировать соответствующие методы. Названия сгенерированных методов будут звучать как ***getИмяПеременной*** и ***setИмяПеременной***.

Аннотация ***@ToString*** перед классом генерирует реализацию метода *toString()*. В генерируемый метод будут добавлены имя класса, названия всех полей и их значений.

Аннотация ***@EqualsAndHashCode*** перед классом генерирует реализации методов *equals()* и *hashCode()*. В генерируемые методы будут добавлены все нестатические поля класса. Чтобы добавить или исключить из генерируемой реализации какие-либо поля, нужно пометить их аннотациями ***@EqualsAndHashCode.Include*** и ***@EqualsAndHashCode.Exclude*** соответственно (при условии отсутствия аннотации на классе).

Аннотация ***@NoArgsConstructor*** генерирует конструктор без параметров. Использование с флагом ***force=true*** установит значения по умолчанию для этих полей.

Аннотация ***@AllArgsConstructor*** генерирует конструктор, включающий все возможные поля.

Аннотация ***@RequiredArgsConstructor*** генерирует конструктор, включающий все *final*-поля или поля, помеченный аннотацией ***@NonNull***.

Аннотация ***@NonNull*** генерирует проверку поля на неинициализированное значение *null*. При проверке возможна генерация исключения *NullPointerException*.

Аннотация ***@Data*** является группировкой сразу пяти аннотаций - *@Getter*, *@Setter*, *@ToString*, *@EqualsAndHashCode* и *@RequiredArgsConstructor*. Она создает полностью **изменяемый** (***muttable***) объект.

Аннотация ***@Value*** является группировкой сразу четырех аннотаций - *@Getter*, *@ToString*, *@EqualsAndHashCode* и *@AllArgsConstructor*. Она создает полностью **неизменяемый** (***immuttable***) объект с *final* и *private* полями.

Аннотация ***@Slf4j*** генерирует объявление переменной *log* и подключение необходимых для нее импортов библиотек логирования *slf4j*.

**Паттерны** – это повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого частного контекста. Существует 22 классических паттерна проектирования. Они делятся на три группы:

* **Порождающие** применяются для гибкого создания объектов без внесения в код лишних зависимостей.
* **Структурные** показывают различные способы построения связей между объектами.
* **Поведенческие** описывают эффективную коммуникацию между объектами.

Если алгоритм – это четкая последовательность действий для получения конкретного результата, то паттерн – это общее высокоуровневое описание задачи (концепция).

**Паттерн *Singleton* (одиночка)** – гарантирует создание только единственного экземпляра класса. Подходит для конфигурации.

**Жадная одиночка** – захватывает ресурсы сразу при старте:

*public class Configuration {*

*private String prop1;*

*private final static Configuration instance = new Configuration();*

*private Configuration() {*

*// read properties from file }*

*public static Configuration getInstance() {*

*return instance; }*

**Ленивая (*Lazy*) одиночка** – захватывает ресурсы при первом обращении (небезопасно в многопоточном приложении):

*public class Configuration {*

*private String prop1;*

*private final static Configuration instance = null;*

*private Configuration() {*

*// read properties from file }*

*public static Configuration getInstance() {*

*if (instance == null) { instance = new Configuration(); }*

*return instance; }*

**Ленивая (*Lazy*) одиночка через *Lombok*** – захватывает ресурсы при первом обращении (безопасно в многопоточном приложении):

*public class Configuration {*

*private String prop1;*

*@Getter(lazy = true)*

*private final static Configuration instance = new Configuration();*

*private Configuration() {*

*// read properties from file }*

**Паттерн *Builder*** (**строитель**) – порождающий паттерн проектирования, который применяется для удобного построения сложного объекта. Его применение имеет следующие преимущества:

* Можно создавать объекты поэтапно.
* Можно использовать один и тот же код для создания различных объектов.
* Код сборки изолируется от основной логики программы.

Можно сгенерировать конструкторы *Builder* через *idea* и вручную дописывать валидацию, либо использовать аннотации *Lombok*.

Аннотация ***@Builder*** на классе позволяет использовать одноименный паттерн.

Аннотация ***@Builder.Default*** на поле переменной позволяет присвоить значение по умолчанию.

Внешне использование паттерна *Builder* похоже на построение стрима.

*Car firstCar = Car.builder() // вызов класса-строителя*

*.color("white") // установка цвета*

*.power(200) // установка мощности*

*.height(1560) // установка высоты*

*.length(4280) // установка длины*

*.weight(1450) // установка ширины*

*.build(); // создание класса Car*

Использование на классе аннотации с флагом ***@Builder(toBuilder = true)*** позволяет использовать на его неизменяемом (***immuttable***) экземпляре метод toBuilder(), который вернет билдер с уже проинициализированными полями на основе исходного объекта. После этого в них можно поменять значения и сконструировать объект.

*final Point point1 = Point.builder().x(0).y(0).build(); // неизменяемый экземпляр*

*final Point point2 = point1.toBuilder().y(3).build(); // метод toBuilder()*

Пример использования билдера в параметризованных тестах:

*private static Stream<Arguments> invalidUsers() {*

*return Stream.of( Arguments.of(User.builder()*

*.name("Test name")*

*.login("ValidTestLogin")*

*.birthday(LocalDate.of(1964, 6, 11))*

*.build(),*

*"Электронная почта не может быть пустой и должна содержать символ @"),*

*Arguments.of(User.builder()*

*.email("")*

*.name("Test name")*

*.login("ValidTestLogin")*

*.birthday(LocalDate.of(1964, 6, 11))*

*.build(),*

*"Электронная почта не может быть пустой и должна содержать символ @")); }*

*@ParameterizedTest*

*@MethodSource("invalidUsers")*

*public void shouldNotAddFilmWithNotValidFields(User user, String message) {*

*ValidationException exception = assertThrows(ValidationException.class, () -> {controller.add(user);});*

*assertEquals(message, exception.getMessage());*

*assertEquals(controller.getAll().size(), 0); }*

**Паттерн *Factory*** (**фабрика**) – выдает те или иные реализации объектов на основании условий.

*interface Human {*

*void scream(); }*

*class Suzi implements Human {*

*@Override*

*public void scream() { System.out.println(“Suzi scream”); } }*

*class French implements Human {*

*@Override*

*public void scream() { System.out.println(“French scream”); } }*

*interface HumanFactory {*

*Human getHuman(String requirements); }*

*class FrenchHumanFactory implements HumanFactory {*

*public Human getHuman(String requirements) {*

*if (requirements.contains(“french”)) return new French();*

*if (requirements.contains(“girl”)) return new Suzi();*

*return null; } }*

Класс, в котором нет ничего кроме объектов зависимостей, принято называть **класс-каталог**, **класс-реестр** или **класс-контекст**. Все зависимости создаются как статические переменные.

*public class Context {*

*static MapService MAP\_SERVICE = new SomeMapService();*

*static LocationService LOCATION\_SERVICE = new GpsService(); }*

**Внедрение зависимостей** (***Dependency injection***) – это подход, в котором зависимости передаются классу извне. Такой принцип программирования считается реализацией **инверсии управления** (***Inversion of Control***, ***IoC***). В связи с этим технология *Spring* для создания и внедрения зависимостей получила название – ***IoC*-контейнер**. *IoC*-контейнер помогает создавать и передавать зависимости автоматически.

***Bean*** (кофейное зерно) – это объекты, которые находятся в контейнере под управлением *Spring*. Для их получения используется метод ***getBean***.

***ApplicationContext*** (контекст приложения) – основной интерфейс *IoC*-контейнера. Его реализации отвечают за хранение информации о классах приложения (в том числе о том, как они создаются, какие у них конструкторы, зависимости и т.д.). При наличии в коде аннотации ***@SpringBootApplication*** *Spring Boot* самостоятельно создает контекст приложения, в других случаях – его создает сам разработчик.

Аннотация ***@Component*** используется для обозначения базового класса.

Аннотация ***@Service*** используется для обозначения класса, в котором происходит обработка логики приложения. С точки зрения *Spring* ничем не отличается от ***@Component***.

Аннотации ***@Component***, ***@(Rest)Controller*** и ***@Service*** позволяют автоматически добавлять классы в контекст. По умолчанию *Spring Boot* проверяет наличие этих аннотаций в том же пакете (включая вложенные пакеты), что и класс, помеченный аннотацией *@SpringBootApplication*. Для изменения пути можно использовать параметр аннотации ***scanBasePackages***.

Аннотация ***@Service*** делает то же самое, что и @Component. Используется для обозначения класса, в котором происходит обработка логики приложения.

Аннотация ***@Qualifier(“name”)*** позволяет присваивать компонентам уникальные имена, а при связывании через *@Autowired* указывать какие компоненты внедрять. Альтернатива – использовать аннотацию ***@Primary*** на компонентах для указания приоритетного, либо создание конструкторов через *Lombok*.

Аннотация ***@Autowired*** указывает на то, что в этом месте произойдет автоматическое внедрение зависимостей контекста через *Spring*. Она может стоять перед конструктором, полем и методом класса.

Способы внедрения зависимостей через аннотацию *@Autowired*:

1. **Напрямую в поля**. Не рекомендован к использованию, т.к. есть возможность перегрузить класс зависимостями вместо конструктора. Главный плюс – краткость. Минусы:

* Нельзя использовать *final*-поля, т.к. для внедрения поле должно быть изменяемым.
* Сложнее тестировать, т.к. поля как правило закрыты модификатором *private*.
* Если бина нет в контейнере, то поле будет равно *null*, но об этом станет известно только при попытке к нему обратиться.

*@Component*

*public class MyCar {*

*@Autowired*

*private Engine engine; }*

1. **Через методы-сеттеры**. Главный плюс – возможность работать с циклическими зависимостями. Минусы:

* Нельзя использовать *final*-поля, т.к. для внедрения поле должно быть изменяемым.
* Если бина нет в контейнере, то поле будет равно *null*, но об этом станет известно только при попытке к нему обратиться.

*@Component*

*public class MyCar {*

*private Engine engine;*

*@Autowired*

*public void setEngine(Engine engine) {*

*this.engine = engine; } }*

1. **Через конструктор**. Рекомендован к использованию как самый надежный способ. Плюсы – возможность использования *final*-полей и отсутствие сложностей с тестированием. Минусы – громоздкий конструктор при большом количестве параметров, конструктор для внедрения зависимостей может быть только один (аннотация *@Autowired* для конструктора не обязательна).

*@Component*

*public class MyCar {*

*private final Engine engine;*

*private final Wheels wheels;*

*private final Radio radio;*

*@Autowired*

*public MyCar(Engine engine, Wheels wheels, Radio radio) {*

*this.engine = engine;*

*this.wheels = wheels;*

*this.radio = radio; } }*

1. **Через *Lombok* и аннотацию *@RequiredArgsConstructor***. Он создаст примерно такой же конструктор, что написан выше.

*@Component*

*@RequiredArgsConstructor*

*public class MyCar {*

*private final Engine engine;*

*private final Wheels wheels;*

*private final Radio radio; }*

1. **Комбинирование разных способов**. Не рекомендован к использованию, т.к. сложен для чтения.

По умолчанию все зависимости являются обязательными. Зависимость может быть **необязательной**, если отсутствие зависимости не нарушает работу программы. В таком случае она несет опциональную функциональность и её можно пометить с помощью параметра аннотации ***required = false***.

*@Component*

*public class MyCar {*

*private final Engine engine;*

*private final SeatHeater seatHeater;*

*@Autowired*

*public MyCar(Engine engine,@Autowired(required = false) SeatHeater seatHeater) {*

*this.engine = engine;*

*this.seatHeater = seatHeater; }*

*public void start() {*

*engine.start();*

*if (seatHeater != null) seatHeater.start(); } }*

**Циклическая зависимость** получается тогда, когда классы зависят одновременно друг от друга. Например, курица и яйцо. Получаем ошибку создания бина, который уже в процессе создания. Частично можно обойти эту ошибку через внедрение с помощью методов-сеттеров.

Обычно наличие циклических зависимостей указывает на то, что требуется рефакторинг.

Часть пути эндпоинта, записанная в фигурных скобках называется **переменной пути**. Она не фиксирована и может изменяться.

Аннотация ***@PathVariable*** используется для обращения к переменной пути. Ею помечается параметр метода контроллера. Если имя переменной пути и имя параметра метода отличаются, то мы можем указать его в аргументе аннотации. В запросе могут использоваться несколько переменных пути или мапа. Параметры метода с этой аннотацией требуются по умолчанию, в противном случае нужно указать параметр аннотации ***(required = false)***или использовать ***java.util.Optional<T>***. Подробнее можно почитать по [ссылке](https://www.baeldung.com/spring-pathvariable).

*@GetMapping("/posts/{postId}")*

*public Optional<Post> findById(@PathVariable(“postId”) int postId) {*

*return posts.stream()*

*.filter(x -> x.getId() == postId)*

*.findFirst(); }*

Для отправки к серверу запроса с параметрами используется **строка запроса**. Аргументы в строке запроса отделяются от пути знаком вопроса **?** и указываются в формате **параметр=значение**. Символ **&** разделяет аргументы.

Аннотация ***@RequestParam*** используется для обращения к аргументу метода контроллера. В запросе могут использоваться несколько переменных пути или мапа. Если имя переменной пути и имя параметра метода отличаются, то мы можем указать его в аргументе аннотации. Параметры метода с этой аннотацией требуются по умолчанию (при отсутствии параметра выбросит исключение), в противном случае нужно указать параметр аннотации ***(required = false)***или использовать ***java.util.Optional<T>***, или установить значение по умолчанию через параметр аннотации ***defaultValue***. Аннотация не умеет читать время и дату. Подробнее о работе с этой аннотацией можно почитать по [ссылке](https://www.baeldung.com/spring-request-param).

Аннотация ***@DateTimeFormat*** используется для обращения к аргументу метода контроллера. С помощью параметра ***pattern*** можно указать формат даты. Подробнее о работе с датами можно почитать по [ссылке](https://www.baeldung.com/spring-date-parameters).

*.../posts/search?author=Bob&date=2020-12-31*

*@GetMapping("/posts/search")*

*public List<Post> searchPosts( @RequestParam(name = “author”) String author,*

*@RequestParam @DateTimeFormat(pattern = "yyyy-MM-dd") LocalDate date) {*

*System.out.println("Ищем посты пользователя с именем " + author + " и опубликованные " + date); }*

Пример постраничного вывода.

@GetMapping("/posts")  
public List<Post> findAll(@RequestParam(value = "page", defaultValue = "0", required = false) Integer page,  
 @RequestParam(value = "size", defaultValue = "10", required = false) Integer size,  
 @RequestParam(value = "sort", defaultValue = "desc", required = false) String sort) {  
 if(!(sort.equals("asc") || sort.equals("desc"))){  
 throw new IllegalArgumentException();  
 }  
 if(page < 0 || size <= 0){  
 throw new IllegalArgumentException();  
 }  
  
 Integer from = page \* size;  
 return postService.findAll(size, from, sort);  
}

public List<Post> findAll(Integer size, Integer from, String sort) {  
 return posts.stream().sorted((p0, p1) -> {  
 int comp = p0.getCreationDate().compareTo(p1.getCreationDate()); //прямой порядок сортировки  
 if(sort.equals("desc")){  
 comp = -1 \* comp; //обратный порядок сортировки  
 }  
 return comp;  
 }).skip(from).limit(size).collect(Collectors.*toList*());  
}

В *Spring Boot Web* содержится библиотека ***Jackson***, в которой есть класс ***ObjectMapper***, умеющий сериализовать и десериализовать объекты и преобразовывать их в бины с помощью аннотаций. С помощью параметра ***produces*** в маппинге можно менять тип тела ответа. Подробнее можно почитать по [ссылке](https://www.baeldung.com/spring-request-response-body).

Аннотация ***@RequestBody*** позволяет преобразовать тело запроса в объект указанного типа (десериализация). По умолчанию используется формат *JSON*.

Аннотация ***@ResponseBody*** позволяет преобразовать объект указанного типа в тело запроса *JSON* (сериализация). Указывается вручную перед методом в контроллере с аннотацией ***@Controller*** и автоматически в контроллере с аннотацией ***@RestController***.

Аннотация ***@ExceptionHandler*** используется перед методами контроллера и отлавливает выбрасываемое исключение. В качестве возвращаемого значения можно передать мапу, т.к. ответ по умолчанию будет сконвертирован в *JSON*.

*@ExceptionHandler // отлавливаем исключение IllegalArgumentException*

*public Map<String, String> handleNegativeCount(final IllegalArgumentException e) {*

*return Map.of("error", "Передан отрицательный параметр count."); }*

Также можно обрабатывать несколько исключений, если они наследуются от одного родителя. Для этого нужно перечислить их в параметрах аннотации в фигурных скобках через запятую, а в аргументах указать родительское исключение.

*@ExceptionHandler({IllegalArgumentException.class, NullPointerException.class})*

*public Map<String, String> handleIncorrectCount(final RuntimeException e) {*

*return Map.of( "error", "Ошибка с параметром count.",*

*"errorMessage", e.getMessage() ); }*

Однако при таком подходе будет возвращаться код **200 ОК**. Изменить это можно несколькими способами:

1. В качестве ответа возвращать экземпляр класса ***ResponseEntity***.

*@ExceptionHandler*

*public ResponseEntity<Map<String, Integer>> handle(final HappinessException e) {*

*return new ResponseEntity<>( Map.of("happinessLevel", e.getHappinessLevel()),*

*HttpStatus.BAD\_REQUEST ); }*

1. Использовать аннотацию ***@ResponseStatus***, а тело ответа не менять.

*@ExceptionHandler*

*@ResponseStatus(HttpStatus.BAD\_REQUEST)*

*public Map<String, String> handleNegativeCount(final IllegalArgumentException e) {*

*return Map.of("error", "Передан отрицательный параметр count."); }*

Частая практика – создание специального объекта для универсального формата ошибки. То есть все исключения отлавливаются и выбрасывают исключение с универсальным форматом.

*public class ErrorResponse {*

*String error;*

*String description;*

*public ErrorResponse(String error, String description) {*

*this.error = error;*

*this.description = description; }*

*// геттеры необходимы, чтобы Spring Boot мог получить значения полей*

*public String getError() {*

*return error; }*

*public String getDescription() {*

*return description; } }*

В Spring Boot есть специальное непроверяемое исключение ***ResponseStatusException***. У него есть три конструктора, которые могут принимать от одного до трех параметров в следующем порядке:

1. ***HttpStatus*** – перечисление обозначающее код ответа.
2. ***String reason*** – текстовое сообщение, которое может передать сервер. Для этого нужно в *application.properties* установить параметр ***server.error.include-message=always***.
3. ***Throwable cause*** – произошедшее ранее исключение.

*@GetMapping("/feed")*

*public Map<String, Integer> feed() {*

*throw new ResponseStatusException(HttpStatus.NOT\_IMPLEMENTED, "Метод /feed ещё не реализован."); }*

Список кодов состояния HTTP можно посмотреть по [ссылке](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%BA%D0%BE%D0%B4%D0%BE%D0%B2_%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D1%8F_HTTP).

Аннотация ***@RestControllerAdvice*** позволяет одинаково обрабатывать все исключения в приложении. Она ставится над отдельным классом, в котором перечисляются обработчики ***@ExceptionHandler***. Эта аннотация совмещает в себе ***@ControllerAdvice*** и ***@ResponseBody***.

Выбрать конкретные классы для обработки можно с помощью параметра аннотации ***assignableTypes***.

*@ControllerAdvice(assignableTypes = CatsInteractionController.class)*

*@ControllerAdvice(assignableTypes = {DogsInteractionController.class, CatsInteractionController.class})*

Выбрать все классы пакета для обработки можно одним из следующих образов.

*@ControllerAdvice("ru.yandex.practicum.controllers")*

*@ControllerAdvice(value = "ru.yandex.practicum.controllers")*

*@ControllerAdvice(basePackages = "ru.yandex.practicum.controllers")*

Про форматирования текста в *Readme.md* можно почитать [тут](https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax#images).

***Structured Query Language, SQL*** (**язык структурированных запросов**) – это непроцедурный язык программирования, который работает практически с любой базой данных. Он является **декларативным** и описывает что, а не как нужно сделать. Он не чувствителен к реестру, но лучше соблюдать стиль.

**Реляционная БД** – это БД, в которой данные представлены в виде связанных таблиц. Столбцы называются **полями, колонками** или **признаками**, строки – **записями**, объекты пересечения полей и записей – **ячейками**.

***Database management system*, СУБД** – это система управления базами данных.



Для корректного написания SQL-запросов можно использовать [*SQLFormater*](https://sqlformat.org/) или следующие правила:

1. Писать операторы в верхнем регистре, чтобы визуально отделить их от ключевых слов.
2. Переносить новое ключевое слово на другую строку, чтобы легче понять какие столбцы будут выгружены.

Чтобы закомментировать строку в SQL используются два дефиса ***--***. Чтобы закомментировать несколько строк используются символы ***/\**** в начале и ***\*/*** в конце.

Типы данных в *PostgreSQL*:

* ***integer*** хранит целые числа от -2147483649 до +2147483647. Его обозначают либо ***int***, либо ***int4*** (занимает 4 байта).
* ***real*** обозначает вещественные числа (дробные). Его обозначают ***float4***.
* ***character*** содержит текст фиксированной длины. Псевдоним – ***char(n)***, где *n* – максимальное количество символов в строке (недостающие символы будут заполнены пробелами). По умолчанию = 1.
* ***character varying*** содержит тексты нефиксированной длины. Псевдоним – ***varchar(n)***, где *n* управляет лимитом строки (можно не указывать).
* ***text*** содержит строку любой длины. Нестандартный тип в *SQL*, используется в некоторых СУБД.
* ***date*** содержит только дату. Обычно используется в международном формате *ISO* *‘YYYY-MM-DD’*.
* ***timestamp*** содержит дату и время. Подразделяется на *timestamp without time zone ‘2004-10-19 10:23:54’* (используется по умолчанию) и *timestamp with time zone ‘2004-10-19 10:23:54+02’*.
* ***boolean*** обозначает логический тип. Содержит два значения *TRUE* ('true', 't', 'yes', 'y', 'on', '1') и *FALSE* (‘false', 'f', 'no', 'n', 'off', '0').

Операторы обращения к БД:

* ***SELECT*** – описывает что выгружать из базы.
* ***FROM*** – описывает откуда именно выгружать данные.
* ***WHERE*** – оператор сравнения, используется как фильтр при выборке (***>, <, >=, <=, =, !=, <>***). Ставят после *FROM*. При сравнении с символьным типом, датой и временем набор символов нужно взять в ***‘***одинарные кавычки***’***. Работает только с изначальной таблицей, но не с результатом агрегирующих функций.
* ***HAVING*** – используется, чтобы получить срез данных группировки (результат агрегирующих функций). Используется после *GROUP BY* (если есть). Например, *HAVING SUM(total) > 41*.
* ***AND, OR, NOT*** – логические операторы, используются для комбинирования условий выборки через оператор *WHERE*. Приоритет операторов в порядке уменьшения – *NOT, AND, OR*. В остальных случаях используют ***(***скобки***)***, действия в которых имеет наивысший приоритет.
* ***IS NULL, IS NOT NULL*** – используются для фильтрации или проверки наличия данных. В *SQL* используется специальное значение ***NULL*** для обозначения отсутствия данных. С *NULL* нельзя сравнивать какое-либо значение.
* ***GROUP BY*** – оператор группировки данных по полю. Ставят после *WHERE* (если отсутствует – после *FROM*). Порядок группировки по нескольким полям не влияет на выдачу. Запрос *SELECT city FROM table GROUP BY city* выгрузит все уникальные значения в поле (аналогично *DISTINCT*).
* ***ORDER BY*** – оператор сортировки. После него можно указать только *LIMIT*. По умолчанию сортирует данные от меньшего к большем. Чтобы изменить порядок сортировки вручную, после названия поля указывают ключевое слово ***DESC*** (по убыванию) или ***ASC*** (по возрастанию). Строки будут отсортированы в лексикографическом порядке. Порядок сортировки по нескольким полям влияет на выдачу.
* ***IN, NOT IN*** – позволяет определить, совпадает ли значение объекта со значением в списке. Например, *WHERE billing\_country IN ('USA', 'United Kingdom', 'Germany')*.
* ***LIMIT*** – указывает количество первых записей, которые нужно выгрузить. Ставят в конце запроса.

Помимо операций выборки данных, *SQL* позволяет манипулировать данными.

1. Поддерживаются математические операции ***+, -, \*, /*** с числами, но не поддерживает операции конкатенации строк.
2. Поддерживаются агрегирующие функции.

* ***SUM(поле)*** – возвращает сумму значений в поле.
* ***AVG(поле)*** – находит среднее арифметическое для значений в поле.
* ***MIN(поле)*** – возвращает минимальное значение в поле.
* ***MAX(поле)*** – возвращает максимальное значение в поле.
* ***COUNT(поле)*** – выводит количество записей в поле без учета пропусков. Можно использовать для всей таблицы (пропуски будут учтены).
* ***ROUND(поле)*** – округляет поле до ближайшего целого числа.

**Первичный ключ** (***primary key, PK***) – это уникальный признак записи. Он обеспечивает уникальность записей и может быть представлен набором любых символов.

**Внешний ключ** (***foreign key, FK***) – это поле, которое отсылает к первичному ключу другой таблицы.

В реляционных БД существует три основных вида связи таблиц:

1. **Один к одному (*one-to-one*)** – одна запись первой таблицы связана только с одной записью другой таблицы. Используется редко, т.к. такие таблицы можно объединить в одну.

Например, запись о сотруднике из таблицы отдела разработки связана с таблицей всего персонала связью *one-to-one*.

1. **Один ко многим (*one-to-many*)** – одной записи первой таблицы соответствует несколько записей другой таблицы. Самый популярный вид связи.

Например, запись о клиенте из таблицы с клиентами связана с несколькими заказами из таблицы заказов связью *one-to-many*.

1. **Многие ко многим (*many-to-many*)** – одной записи первой таблицы соответствует несколько записей другой таблицы и наоборот, одной записи другой таблицы соответствует несколько записей первой таблицы.

Например, отношение таблиц с вкладчиками и банками. Один человек может открыть несколько вкладов в разных банках, а один банк может открывать вклады разным людям.

Реализовать такую связь бывает сложно и неудобно, поэтому используются **соединительные таблицы**. В нашем примере это будет таблица с *id* вкладчиков и *id* банков реализующая связь *one-to-many*.

На *ER*-диаграмме типы связей отображены таким образом:



Пример связывания таблиц:



Ключевое слово ***DISTINCT*** возвращает уникальные значения. Обычно используется после *SELECT*. Если перечислить несколько полей после *DISTINCT*, запрос выведет все уникальные комбинации значений в этих полях. Может сочетаться с агрегирующими функциями.

Функция ***CAST*** приводит значение к заданному типу данных. Например, *CAST(invoice\_date AS DATE)* конвертирует значение поля *invoice\_date* к типу *date*.

Функция ***EXTRACT*** позволяет извлечь значение из данных. Например, можно извлечь год с помощью такой записи *EXTRACT(YEAR FROM CAST(invoice\_date AS DATE))*.

Предикат ***LIKE*** ищет нужную строку в текстовых значениях поля. Символ ***%*** соответствует любому количеству символов. Символ ***\_*** соответствует одному символу. Например, отфильтруем записи, содержащие слово *Epic* – *WHERE descrition LIKE ‘%Epic%’*.

**Псевдоним (*alias*)** – временно название, которое присваивают в запросе таблице или полю. При этом в исходной БД названия не меняются. К псевдонимам нельзя обращаться в *WHERE* или *HAVING* (во многих и в *GROUP BY*), потому что сначала данные выбираются по условию, а только потом полям назначают псевдонимы.

Задать псевдоним можно двумя способами – через пробел или ***AS***:

*SELECT EXTRACT(YEAR FROM CAST(invoice\_date AS DATE)) AS year\_of\_purchase*

*SELECT EXTRACT(YEAR FROM CAST(invoice\_date AS DATE)) year\_of\_purchase*

При использовании псевдонимов таблиц можно указывать имена полей через точку после псевдонима. При работе с несколькими таблицами в одном запросе полезно указывать вместе с полем и таблицу-источник.

*FROM invoice AS i*

*MIN(i.total)*

Операторы JOIN объединяют таблицы “по горизонтали”, а операторы UNION – “по вертикали”.



Оператор ***INNER JOIN*** (внутреннее объединение), или кратко ***JOIN***, предполагает объединение по “внутренней области”, общей для двух таблиц. Например, объединим таблицы по актеру.

Оператор ***LEFT OUTER JOIN*** (левое внешнее объединение), или кратко ***LEFT JOIN***, предполагает, что в результат слияния войдут все записи из левой таблицы. Записи из правой таблицы сохранятся только в том случае, если значения в нужном поле совпадают со значениями в левой таблице. Все из левой плюс общее из правой.

Оператор ***RIGHT OUTER JOIN*** (правое внешнее объединение), или кратко ***RIGHT JOIN***, предполагает, что в результат слияния войдут все записи из правой таблицы. Записи из левой таблицы сохранятся только в том случае, если значения в нужном поле совпадают со значениями в правой таблице. Все из правой плюс общее из левой.

Оператор ***FULL OUTER JOIN*** (полное внешнее объединение), или кратко ***FULL JOIN***, объединяет все данные из левой и правой таблиц. Если не нашлось совпадений, на месте значения будет *NULL*.



Объединять таблицы можно не только по полям, которые являются внешними ключами, но и по другим полям, если типы этих полей совпадают.

Оператор ***UNION*** предполагает склейку двух таблиц с одинаковым количеством и типом полей. Абсолютные дубликаты в итоговую таблицу не входят. Используется между *SELECT*.

Оператор ***UNION ALL*** предполагает склейку двух таблиц с одинаковым количеством и типом полей. Абсолютные дубликаты в итоговую таблицу входят. Используется между *SELECT*.

Таблицу можно присоединять саму к себе. Например, для выстраивания иерархии сотрудников отдела.



**Подзапрос**, или **вложенный запрос** – это обычный запрос, который используется после *FROM* или *WHERE*, т.е. полученную таблицу с данными можно использовать вместо условия. Преимущества – сначала идет отбор данных, а потом объединение таблиц, что позволяет экономить время.

**Нормализация** – это метод проектирования БД, который позволяет привести БД к минимальной избыточности. Другими словами – это процесс удаления избыточных данных. Она нужна для устранения аномалий, повышения производительности и повышения удобства управления данными (где-то до *3NF*).

Степень нормализации градируется по следующим формам (в порядке увеличения нормальности):

1. **Ненормализованная форма или нулевая нормальная форма (*UNF*)**:

* БД приведена к табличному виду так, чтобы он отвечал базовым принципам реляционной теории (строки и столбцы не должны быть пронумерованы, их порядок не должен иметь значения).



1. **Первая нормальная форма (*1NF*)**:

* В таблице не должно быть дублирующих строк.
* В каждой ячейке таблицы хранится атомарное (одно не составное) значение.
* В столбце хранятся данные одного типа.
* Отсутствуют массивы и списки в любом виде.



1. **Вторая нормальная форма (*2NF*)**:

* Таблица должна иметь ключ, по которому можно идентифицировать каждую строку.
* Все не ключевые столбцы таблицы должны зависеть от полного ключа (если он составной). Если какой-то столбец зависит только от части составного ключа, то БД не находится в *2NF*.

Пример с составным ключом (потребует декомпозиции и соединительной таблицы):

1. **Третья нормальная форма (*3NF*)**:

* В таблицах должна отсутствовать **транзитивная зависимость** (когда не ключевые столбцы зависят от значений других не ключевых столбцов).

1. **Нормальная форма Бойса-Кодда (*BCNF*)**:

* Ключевые столбцы составного ключа не должны зависеть от не ключевых столбцов.

1. **Четвертая нормальная форма (*4NF*)**:

* В таблице не должно быть многозначных зависимостей (когда два столбца никак не связаны друг с другом, но оба зависят от третьего столбца).

1. **Пятая нормальная форма (*5NF*)**:

* В таблице каждая нетривиальная зависимость соединения должна определяться потенциальным ключом этой таблицы.

1. **Доменно-ключевая нормальная форма (*DKNF*)**:

* Каждое наложенное ограничение на таблицу должно являться логическим следствием ограничений доменов и ограничений ключей, которые накладываются на данную таблицу.

**Ограничение домена** – это ограничение, предписывающее использование для определенного столбца значений только из некоторого заданного домена (набора значений).

**Ограничение ключа** – это ограничение, утверждающее, что некоторый столбец или их комбинация представляет собой потенциальный ключ.

1. **Шестая нормальная форма (*6NF*)**:

* Таблица должна удовлетворять всем нетривиальным зависимостям соединения.

Приведение БД к какой-то конкретной нормальной форме требует, чтобы эта база уже находилась в предыдущей нормальной форме. БД считается нормализованной, если она находится как минимум в третьей нормальной форме.

**CRUD** (Create/Read/Update/Delete) – стандартная классификация функций по манипуляции данными. **CRUD приложение** – это приложение, реализующее весь этот функционал. Рассмотрим общепринятую конвенцию именования запросов URL на примере сущности *Post*:

|  |  |  |
| --- | --- | --- |
| **HTTP метод** | **URL** | **Действие** |
| GET | /posts | Получаем все записи (READ) |
| POST | /posts | Создаем новую запись (CREATE) |
| GET | /posts/new | HTML форма создания записи |
| GET | /posts/:id/edit | HTML форма редактирования записи |
| GET | /posts/:id | Получаем одну запись (READ) |
| PATCH | /posts/:id | Обновляем запись (UPDATE) |
| DELETE | /posts/:id | Удаляем запись (DELETE) |

Создать новую пустую таблицу можно с помощью оператора ***CREATE TABLE***. Владельцем таблицы будет пользователь, выполнивший команду.

Свойство ***GENERATED BY DEFAULT AS IDENTITY*** означает, что столбец является столбцом идентификации с автоинкрементом.

Задавать ограничения ***CONSTRANT*** (например, *NOT NULL*, *PRIMARY KEY*) можно индивидуально на каждый столбец либо на всю таблицу сразу. Определяя ограничения на уровне таблиц, можно задать ему любое имя.

Оператор ***CHECK*** задает условия проверки.

*CREATE TABLE films ( id INTEGER GENERATED BY DEFAULT AS IDENTITY,*

*title varchar(40) NOT NULL,*

*produced integer NOT NULL,*

*date\_prod date,*

*kind varchar(10),*

*len\_min integer,*

*CONSTRAINT films\_pk PRIMARY KEY (id),*

*CONSTRAINT constr\_example CHECK (len\_min > 100 AND title <> '') );*

Свойство ***DEFAULT*** позволяет задавать значения по умолчанию для столбцов.

*name varchar(40) DEFAULT 'unknown'*

Свойство ***REFERENCES*** позволяет указать, что столбец является вторичным ключом для какой-то другой таблицы. После него указывается имя таблицы и связанный столбец. Все таблицы, на которые мы ссылаемся, к этому моменту должны быть созданы.

При удалении полей из связанных столбцов можно использовать ограничения:

* ***RESTRICT*** – запрещает удаление записи, на которую есть ссылка
* ***CASCADE*** – удаляет все связанные записи
* ***NO ACTION*** – выдаст ошибку. Используется по умолчанию.
* ***SET NULL*** и ***SET DEFAULT*** – присваивает значение NULL или значение по умолчанию

*CREATE TABLE favorite\_films (*

*id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,*

*user\_id INTEGER REFERENCES users (id) ON DELETE RESTRICT,*

*film\_id INTEGER REFERENCES films (id) ON DELETE CASCADE );*

Удалить таблицу и все связанные с ней объекты можно с помощью оператора ***DROP TABLE***. Если на удаляемую таблицу ссылается другая таблица, то удаление не сработает, т.к. по умолчанию действует режим *RESTRICT*. Если явно использовать режим *CASCADE*, то таблица, которая ссылается на данные из текущей таблицы, будет полностью удалена.

*DROP TABLE films, users CASCADE;*

Оператор ***IF EXISTS*** позволяет проверить наличие таблицы перед операцией над ней.

*DROP TABLE IF EXISTS films;*

Изменить существующую таблицу можно с помощью оператора ***ALTER TABLE***. С его помощью можно добавлять и удалять столбцы и ограничения, а также переименовывать столбцы и таблицы. Новые столбцы заполняются *NULL* или заданными значениями по умолчанию.

*ALTER TABLE products ADD COLUMN description text;*

*ALTER TABLE products DROP COLUMN description;*

При добавлении ограничения указывается его имя, через которое потом можно к нему обратиться, а также выполняется автоматическая проверка уже внесенных данных. Примеры редактирования ограничений:

*// установить ограничение на значение имени (не должно совпадать с 'что-то странное')*

*ALTER TABLE products ADD CHECK (name <> 'что-то странное');*

*// установить/удалить ограничение на уникальность имени*

*ALTER TABLE products ADD CONSTRAINT uq\_products\_name UNIQUE (name);*

*ALTER TABLE products DROP CONSTRAINT uq\_products\_name UNIQUE (name);*

*// поле some\_id должно быть внешним ключом к таблице another\_table*

*ALTER TABLE example ADD CONSTRAINT fk\_example\_to\_another\_table FOREIGN KEY (some\_id) REFERENCES another\_table;*

*// установить/удалить ограничение на категорию NOT NULL*

*ALTER TABLE products ALTER COLUMN category SET NOT NULL;*

*ALTER TABLE products ALTER COLUMN category DROP NOT NULL;*

*// установить/удалить столбцу новое значение по умолчанию*

*ALTER TABLE products ALTER COLUMN price SET DEFAULT 100;*

*ALTER TABLE products ALTER COLUMN price DROP DEFAULT;*

*// преобразовать столбец в другой тип данных*

*ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);*

*// переименовать столбец и таблицу*

*ALTER TABLE products RENAME COLUMN name TO title;*

*ALTER TABLE products RENAME TO items;*

**Для добавления новой записи** в таблицу используется оператор ***INSERT***. Далее указываются имя таблицы и столбцов (в любом порядке), а сами значения перечисляются после свойства ***VALUES***. Если список столбцов опущен, то он генерируется автоматически (не рекомендуется).

*INSERT INTO clients (name, phone, email) VALUES ('Кое-кто', '80932334444', 'koe\_kto@mail.ru'),*

*('Еще-кто-то', '80932334445', 'kto@mail.ru');*

Скрыть потенциальную ошибку при добавлении можно с помощью свойства ***ON CONFLICT***. Например:

*INSERT INTO products (id, name, price, category)*

*VALUES (1, 'Шампунь 2', 1200, 'косметика')*

*ON CONFLICT DO NOTHING;*

Вместо ***DO NOTHING*** можно написать ***DO UPDATE*** – она гарантирует добавление или обновление данных. Для обращения к значениям, изначально предлагаемым для добавления, используется специальная таблица ***excluded***.

*INSERT INTO clients (id, name, phone, email)*

*VALUES (9, 'Анна Васильевна Орешкина', '805565423422', 'anna@mail.com')*

*ON CONFLICT (id) DO UPDATE SET name = EXCLUDED.name, phone = EXCLUDED.phone, email = EXCLUDED.email;*

Наполнить таблицу выборкой данных на основании другой таблицы можно с помощью ***подзапросов*** внутри *INSERT*. При этом свойство ***VALUES*** не используется. Например:

*INSERT INTO old\_orders (id, client\_id, date, status, address)*

*SELECT id, client\_id, date, status, address*

*FROM orders*

*WHERE date < '2020-01-01';*

**Для обновления значений полей записей** в таблице используется оператор ***UPDATE***. После команды ***SET*** указываются поля с новыми значениями. Также можно использовать фильтры для массовой установки значений. Этот оператор также поддерживает подзапросы.

*UPDATE clients*

*SET phone = 85553332211*

*WHERE name = 'Иванов Иван Иванович';*

С помощью свойства ***FROM***можно объединить несколько таблиц, но целевая строка не должна соединяться с более чем одной строкой из других таблиц.

*UPDATE positions*

*SET amount = amount + 1*

*FROM orders*

*WHERE positions\_.order\_id = orders.id*

*AND orders.client\_id = ( SELECT id*

*FROM clients*

*WHERE name = 'Бук Василий Петрович' );*

**Для удаления записей** из таблицы используется оператор ***DELETE***. Этот оператор также поддерживает подзапросы. Использовать оператор для параллельного удаления из нескольких таблиц запрещено.

*DELETE FROM clients WHERE id = 111;*

**Дамп БД** – это текстовые файлы в формате *SQL* и запросы внутри них. С его помощью можно восстановить БД с нуля.

**Транзакция** — способ выполнения запросов, при котором гарантируется, что в случае ошибки все сделанные в рамках транзакции изменения будут отменены.

***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; } }*

***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.

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);*

Аннотация ***@SpringBootTest*** позволяет запустить целиком все приложение при тестировании.

Аннотация ***@AutoConfigureTestDatabase*** создает тестовую БД вместе основной.

Аннотация ***@RequiredArgsConstructor(onConstructor\_ = @Autowired)*** указывает, что конструктор, созданный через *Lombok*, сможет получать зависимости через механизм *Autowired*.

Аннотация ***@DirtiesContext(classMode = DirtiesContext.ClassMode.AFTER\_EACH\_TEST\_METHOD)*** очищает бины и БД после каждого теста.

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.

Методология разработки *CI/CD* подразумевает непрерывную интеграцию (***Continuous Integration***) и непрерывную доставку (***Continuous Delivery***). Это набор принципов и практик, которые предназначены для повышения удобства и надёжности развёртывания изменений программного обеспечения или продукта.

Преимущества *Spring Boot* по сравнению с чистым *Spring Framework*:

1. **Контроль зависимостей.**

Для контроля зависимостей используются **стартеры** – специальные наборы внешних библиотек для реализации какой-либо функциональности. Они представляют собой *pom*-файл с перечнем необходимых зависимостей.

Плюс стартеров в том, что при их подключении не нужно проверять версию каждой библиотеки на совместимость с другими (это делают разработчики стартеров). Сверять совместимости версии стартера с версией *Spring Boot* необязательно, если проект наследуется от специального стартера ***spring-boot-starter-parent*** – обновляя версию этого стартера, обновляются все остальные стартеры и зависимости.

1. **Упрощенное развертывание приложения** на серверах и других устройствах (*deployment*).

Фреймворк предоставляет специальный плагин для системы сборки (для *Maven* это ***spring-boot-maven-plugin***). Плагин запаковывает все *jar*-файлы используемых зависимостей в один *jar*-файл приложения, а также модифицирует стандартный механизм запуска так, чтобы запакованные зависимости можно было использовать при запуске. Подробнее о технологии Executable Jar можно почитать [тут](https://docs.spring.io/spring-boot/docs/current/reference/html/executable-jar.html).

1. **Автоматическое конфигурирование.**

В основе автоконфигурации лежит *IoC*-контейнер (контекст приложения) и управление бинами. Цель – осуществить настройку наиболее оптимальным образом, чтобы разработчику требовалось минимум действий для запуска и работы над *Spring*-приложением. А именно:

* Включает транзитивную зависимость *spring-boot-autoconfigure*



* *Jar*-файл этой зависимости содержит [набор классов](https://github.com/spring-projects/spring-boot/tree/main/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure), где предусмотрена программная настройка различных технологий. Например, *HTTP/JDBC/JSON*.
* *Spring* находит классы с программными настройками и добавляет информацию о них в *IoC*-контейнер.
* Когда все нужные классы добавлены в *IoC*-контейнер, *Spring* создает их экземпляры, тем самым запуская код, который настраивает нужные технологии.

**Сервлет** – это класс, который принимает запрос и выдает ответ (контроллеры в *Spring* используют их как основу). Базовым классом для сервлетов является ***HttpServlet***.

Параметр ***request*** содержит в себе заголовки, параметры и другие данные *HTTP*-запроса.

Параметр ***response*** – это заготовка для ответа, которую мы заполняем по мере необходимости.

Методы ***doGet***, ***doPost***, ***doPut*** обрабатывают соответствующие *HTTP*-запросы.

Метод ***getParameter*** извлекает значение соответствующего параметра из запроса.

*public class SumServlet extends HttpServlet {*

*@Override*

*protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {*

*int a = Integer.parseInt(req.getParameter("a"));*

*int b = Integer.parseInt(req.getParameter("b"));*

*int sum = a + b;*

*resp.getWriter().print(sum); } }*

По аналогии с *IoC*-контейнером существует **контейнер сервлетов** – программа, которая открывает сетевой порт, принимает соединения, парсит протокол *HTTP*, распределяет работу по ядрам процессора и вызывает методы сервлетов. В одном контейнере сервлетов может одновременно работать несколько сервлетов, которые обрабатывают каждый свой тип *HTTP*-запросов. Самым популярным является ***Apache Tomcat***.

Приложения, состоящие из сервлетов, могут не включать “точку входа” (класс с методом *main*). Контейнер сам настраивается и запускает сервлеты с помощью инверсии контроля. Современные контейнеры сервлетов являются встроенными в приложение, а также содержат методы *main*, *start* и *addServlet*.

*Spring Framework* одинаково может работать и через подкладывание сервлетов на сервер, и со встроенным контейнером. *Spring Boot* же по умолчанию настаивает на подходе со встроенным контейнером, но оставляет возможность использовать вариант с отдельным контейнером.

Обычно *Spring* использует один **сервлет-диспетчер** ***DispatcherServlet***, который принимает на вход все запросы к приложению и перенаправляет их в нужные контроллеры либо выдает ошибку. При запуске приложения на *Spring Boot* автоматически собирается информация обо всех классах с аннотациями *@Controller* и *@RestController* и их методах с аннотациями *@RequestMapping*, *@GetMapping* и другими, а затем сформированный контекст передается в сервлет-диспетчер. В случае со *Spring Framework* за настройку контекста отвечает разработчик.

***ApplicationContext*** – это множество классов или бинов (основной интерфейс *IoC*-контейнера), которые задействованы в приложении. Обычно они отмечены аннотациями *@Component*, *@Service*, *@RestController* и другими.

***WebApplicationContext*** – это интерфейс, расширяющий *ApplicationContext* и содержащий метод для получения *ServletContext*.

*public interface WebApplicationContext extends ApplicationContext {*

*// ... полезные константы ...*

*ServletContext getServletContext(); }*

***ServletContext*** – это класс, который хранит общие настройки и методы для всех сервлетов. С его помощью каждый сервлет может взаимодействовать с контейнером сервлетов. Объект класса *ServletContext* создается при запуске контейнера сервлетов и является общим для всех сервлетов приложения. Например, его можно использовать так:

* метод ***ServletContext.getInitParameter(name)*** позволяет получить настройки приложения из файла конфигурации.
* методы ***ServletContext.setAttribute(name, value)*** и ***ServletContext.getAttribute(name)*** позволяют передать какие-то значения другому сервлету.
* метод ***ServletContext.log(message)*** позволяет что-нибудь залогировать.

Конструктор класса *DispatcherServlet* принимает на вход одну из реализаций *WebApplicationContext*. В большинстве случаев подходит ***AnnotationConfigWebApplicationContext***, которая поддерживает создание бинов с помощью аннотаций *@Component*, *@Service*, *@RestController* и другими. В более старых проектах можно встретить реализацию ***XmlWebApplicationContext***, которая вместо аннотаций ожидает *XML*-конфигурацию, где указаны все бины.

**Рассмотрим пример создания веб-приложения на *Spring Framework*.**

* Для начала создадим новый проект с использованием *Maven* и добавим следующие зависимости:

*<dependencies>*

*<!--встроенный контейнер сервлетов-->*

*<dependency>*

*<groupId>org.apache.tomcat.embed</groupId>*

*<artifactId>tomcat-embed-core</artifactId>*

*<version>9.0.56</version>*

*</dependency>*

*<!--диспетчер сервлетов и контроллеры-->*

*<dependency>*

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

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

*<version>5.3.15</version>*

*</dependency>*

*</dependencies>*

* Создадим тестовый контроллер:

*@RestController*

*public class TestController {*

*@GetMapping("/test")*

*public String test() {*

*return "OK from controller!"; } }*

* Запускаем контейнер сервлетов, создаем диспетчер сервлетов и *Spring*-контекст для поиска бинов.

*public class LaterApplication {*

*private static final int PORT = 8080;*

*public static void main(String[] args) throws LifecycleException {*

*// создадим контейнер сервлетов и сетевой порт*

*Tomcat tomcat = new Tomcat();*

*tomcat.getConnector().setPort(PORT);*

*// укажем путь до приложения пустым -* [*http://localhost:8080/*](http://localhost:8080/)

*Context tomcatContext = tomcat.addContext("", null);*

*// создадим реализацию WebApplicationContext*

*AnnotationConfigWebApplicationContext applicationContext =*

*new AnnotationConfigWebApplicationContext();*

*applicationContext.setServletContext(tomcatContext.getServletContext());*

*// ищем бины и загружаем Spring-контекст*

*applicationContext.scan("ru.practicum");*

*applicationContext.refresh();*

*// добавляем диспетчер запросов*

*DispatcherServlet dispatcherServlet = new DispatcherServlet(applicationContext);*

*// класс Wrapper позволяет задать дополнительные настройки для сервлета*

*Wrapper dispatcherWrapper = Tomcat.addServlet(tomcatContext, "dispatcher", dispatcherServlet);*

*// сервлет будет обрабатывать все пути, начиная с корневого*

*dispatcherWrapper.addMapping("/");*

*// зададим инициализацию сразу при запуске контейнера*

*dispatcherWrapper.setLoadOnStartup(1);*

*tomcat.start(); } }*

* Теперь перейдя на <http://localhost:8080/test> можно получить сообщение “OK from controller!”.

**Добавим в приложение поддержку *JSON*.**

* Сначала добавим следующие зависимости:

*<!--библиотека конвертации в JSON-->*

*<dependency>*

*<groupId>com.fasterxml.jackson.core</groupId>*

*<artifactId>jackson-databind</artifactId>*

*<version>2.13.3</version>*

*</dependency>*

* Создадим пустой класс с аннотациями ***@EnableWebMvc*** и ***@Configuration***. После того как *Sping Framework* считал эту аннотацию, он сам создает дополнительные бины, в том числе с поддержкой *JSON*, а *@EnableWebMvc* просто импортирует эту конфигурацию.

*// помечаем класс как java-config для контекста приложения*

*@Configuration*

*// импортируем дополнительную конфигурацию для веб-приложений  
@EnableWebMvc   
public class WebConfig { }*

* Теперь перейдя на <http://localhost:8080/users> можно получить список фейковых пользователей в формате *JSON*.

**Добавим в приложение поддержку внешнего файла настроек.**

* Создадим пустой класс с аннотацией ***@PropertySource*** и ***@Configuration***. Наличие нескольких аннотаций *@PropertySource* возможно, но в таком случае последние настройки (нижние) перекроют настройки из более ранних (верхних). Это дает возможность задать настройки по умолчанию, а затем переопределить их в других файлах. Параметр ***ignoreResourceNotFound***означает, что возможное отсутствие файла не является ошибкой.

*// помечаем класс как java-config для контекста приложения*

*@Configuration*

*// classpath говорит, что файл нужно искать в самом приложении*

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

*// пример поиска в каталоге для Windows*

*@PropertySource(value = "file:C:\\myapp\\myapp.properties", ignoreResourceNotFound = true)*

*// пример поиска в каталоге для Unix-like*

*@PropertySource(value = "file:/etc/myapp/myapp.properties", ignoreResourceNotFound = true)*

*public class AppConfig { }*

* Создадим файл application.properties и добавим в него значение:

*message=Hello world, from config!*

* Для загрузки значения message нужно использовать следующую конструкцию:

*@RestController*

*@RequestMapping(“/hello”)*

*public class HelloMessageComponent {*

*private final String message;*

*@Autowired*

*public HelloMessageComponent(@Value("${message:Hello!}") String message) {*

*this.message = message; }*

*@GetMapping*

*public String getHelloMessage() {*

*return message; } }*

Также у выражений в ***@Value*** есть полезная возможность задать значение по умолчанию — *@Value("${key:Default value}")*. Если ключ *key* есть в настройках, будет использовано значение из настроек, а если нет, то будет использовано значение “*Default value*”.

* Теперь перейдя на <http://localhost:8080/hello> можно получить сообщение из настройки. Если удалить строку *message* из настроек, то выведется просто *Hello!*.

**Добавим в приложение поддержку логирования.**

* В *Spring Boot* библиотеки ***slf4j*** и ***logback-classic*** добавляются при подключении почти любого стартера, в том числе ***spring-boot-stater-web***. В *Spring Framework* нужно вручную добавить зависимость:

*<!--логирование-->*

*<dependency>*

*<groupId>ch.qos.logback</groupId>*

*<artifactId>logback-classic</artifactId>*

*<version>1.2.11</version>*

*</dependency>*

* Создать экземпляр логгера вручную либо через аннотацию ***@Slf4j***:

*private static final Logger log = LoggerFactory.getLogger(ExampleService.class);*

Аннотация ***@RequestHeader*** позволяет вытащить значение из заголовка *HTTP*-запроса и передать его в параметр *Long userId*. При этом *X-Later-User-Id* – это название заголовка. Например, тут может передаваться *ID* пользователя.

*public void deleteItem(@RequestHeader("X-Later-User-Id") Long userId, @PathVariable Long itemId) { }*

Аннотация ***@Scope*** позволяет задать область видимости бина. Существует несколько видов области видимости и способов их задания перед классом с аннотацией бина (например, *@Component*):

* ***Singleton*** – создает и возвращает на все запросы один и тот же бин. Используется по умолчанию, создается сразу при запуске приложения и получении *ApplicationContext*.

*@Scope(“ConfigurableBeanFactory.SCOPE\_SINGLETON”)*

*@Scope(“singleton”)*

* ***Prototype*** – создает и возвращает на каждый запрос новый бин.

*@Scope(“ConfigurableBeanFactory.SCOPE\_PROTOTYPE”)*

*@Scope(“prototype”)*

* ***Request*** – создает бин на время одного HTTP запроса.

*@Scope(value = WebApplicationContext.SCOPE\_REQUEST, proxyMode = ScopedProxyMode.TARGET\_CLASS)*

*@RequestScope*

* ***Session*** – создает бин на время одной HTTP сессии.

*@Scope(value = WebApplicationContext.SCOPE\_SESSION, proxyMode = ScopedProxyMode.TARGET\_CLASS)*

*@SessionScope*

* ***Application*** – создает бин на время жизненного цикла *ServletContext*.

*@Scope(value = WebApplicationContext.SCOPE\_APPLICATION, proxyMode = ScopedProxyMode.TARGET\_CLASS)*

*@ApplicationScope*

* ***Websocket*** – создает бин на время одной WebSocket сессии.

*@Scope(scopeName = “websocket”, proxyMode = ScopedProxyMode.TARGET\_CLASS)*

При инжекте *Prototype* бина в *Singleton* бин, *Prototype* бин создастся только один раз. Чтобы обойти это ограничение можно использовать аннотацию **@LookUp** перед *public* методом создания *prototype* бинов, которая превращает метод в фабричный.

*@LookUp*

*public PrototypeBean getPrototypeBean() {*

*return null; } // Spring сам переопределит этот метод под фабричный*

Аннотации ***@PostConstruct*** и ***@PreDestroy*** позволяют указать какие методы будут использованы на этапах инициализации и уничтожения бина. Количество методов с такими аннотациями неограничен. У этих методов может быть любые модификатор доступа, тип возвращаемого значения, название метода и не должно быть аргументов.

Сначала запускается конструктор класса, потом метод с *@PostConstruct* и затем метод с *@PreDestroy*. *@PreDestriy* не работает для *prototype scope*.

Библиотека ***MapStruct*** позволяет автоматически генерировать методы преобразования одного объекта в другой (мапперы).

Сначала нужно подключить в проект зависимости:

*<dependency>*

*<groupId>org.mapstruct</groupId>*

*<artifactId>mapstruct</artifactId>*

*<version>1.5.3.Final</version>*

*</dependency>*

*<dependency>*

*<groupId>org.mapstruct</groupId>*

*<artifactId>mapstruct-processor</artifactId>*

*<version>1.5.3.Final</version>*

*</dependency>*

Затем создать интерфейс с аннотацией ***@Mapper***, в котором будут находиться специально именованные методы. Для использования в *Spring* или *CDI* дополнительно указывается параметр со значением *spring* или *cdi*. Затем добавить маппер в нужный класс.

*@Mapper(componentModel = “spring”)*

*public interface SourceDestinationMapper {*

*Destination sourceToDestination(Source source);*

*Source destinationToSource(Destination destination); }*

Аннотация ***@Mapping*** позволяет сопоставить исходное и целевое поле при маппинге. Указывается перед методом в любом количестве.

*@Mapper*

*public interface UserMapper {*

*@Mapping(target=”userId”, source=”entity.id”)*

*@Mapping(target=”name”, source=”entity.name”)*

*UserDto userToUserDto(User entity); }*

Также библиотека может преобразовывать объекты с конвертацией полей (например, *String* и *Date*). Подробнее о работе библиотеки можно почитать [тут](https://www.baeldung.com/mapstruct).

***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.

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.

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

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

Аннотация ***@ManyToMany*** создает связь многие-ко-многим для аннотируемого элемента.

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 сущности. Значение ***LAZY*** не загружает дочерние сущности. Подробнее можно почитать [тут](https://javarush.com/quests/lectures/questhibernate.level14.lecture01).

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 cb = entityManager.getCriteriaBuilder();*

*CriteriaQuery<User> cr = cb.createQuery(User.class);*

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

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

*List<User> foundUsers = entityManager.createQuery(cr).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*.

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 прочитаны ею ранее. 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.

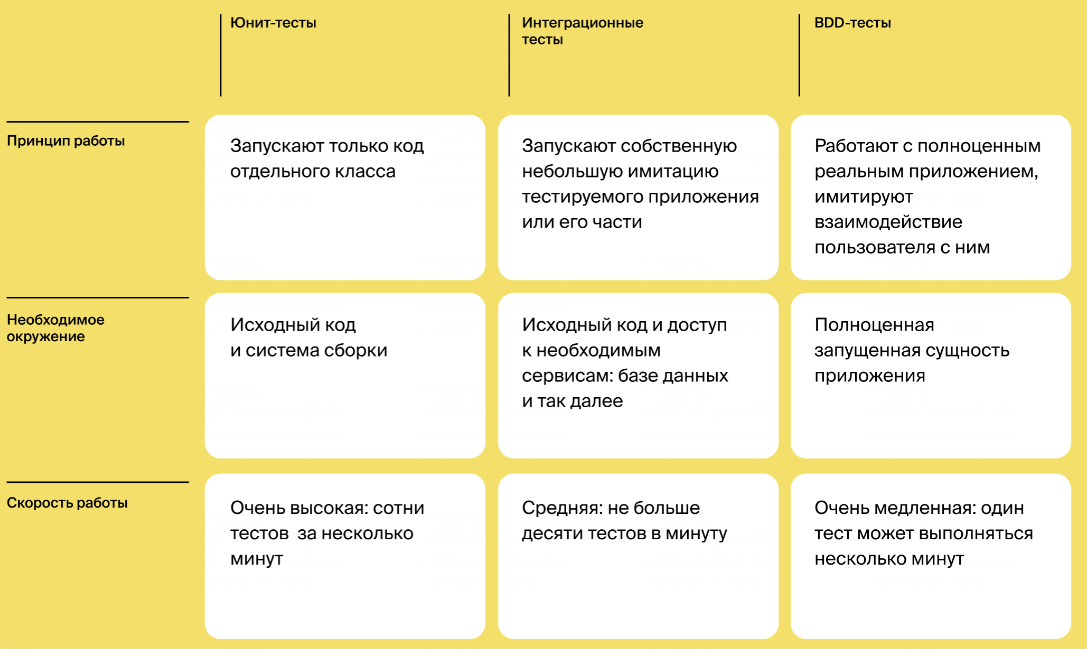
*@RepositoryRestResource(path = "people")*

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

*@RestResource(path = "emails")*

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

Для тестирования отдельных классов и методов используются **юнит тесты**. Для тестирования взаимодействия нескольких классов между собой или взаимодействия классов приложения с внешним окружением (БД) используются **интеграционные тесты**. Для тестирования сценария работы пользователя используются ***BDD*-тесты**.



**Мок** – простой объект-заглушка, к которому можно обращаться, вызывать методы и получать результаты. Например, заменим класс *AuthorService* и перепишем его метод, вернув нужную нам строку:

*class MockAuthorService extends AuthorService {*

*@Override*

*public String getAuthorDescription(int authorId) throws SQLException {*

*return "знаменитый русский писатель"; } }*

Теперь можно написать юнит-тест:

*@Test*

*void testCreateBookDescription() {*

*BookService bookService = new BookService();*

*AuthorService mockAuthorService = new MockAuthorService();*

*bookService.setAuthorService(mockAuthorService);*

*String bookDescription = bookService.createBookDescription("Война и мир", 1898, 5, "Л.Н.Толстой");*

*Assertions.assertEquals("Война и мир, 1898 автор Л.Н.Толстой, знаменитый русский писатель",*

*bookDescription);}*

Для такого мок-тестирования лучше использовать библиотеку ***Mockito***, которая уже включена в стартер ***spring-boot-starter-test***. Перепишем пример выше:

*@Test*

*void testCreateBookDescriptionWithMock() throws SQLException {*

*BookService bookService = new BookService();*

*AuthorService mockAuthorService = Mockito.mock(AuthorService.class); // класс для мока*

*bookService.setAuthorService(mockAuthorService);*

*Mockito // логика уже внутри самого теста*

*.when(mockAuthorService.getAuthorDescription(Mockito.anyInt()))*

*.thenReturn("знаменитый русский писатель");*

*String bookDescription = bookService.createBookDescription("Война и мир", 1898, 5, "Л.Н.Толстой");*

*Assertions.assertEquals("Война и мир, 1898 автор Л.Н.Толстой, знаменитый русский писатель",*

*bookDescription); }*

Метод ***Mockito.when()*** позволяет определить условие, на которое будет реагировать мок. Например, вызывается определенный метод, мокируемого класса, с заданным значением аргументов. Кроме конкретных значений аргументов, можно использовать статические функции *Mockito* из семейства ***anyType()*** – *anyInt()*, *anyString()* и т.д. Такие проверки на значения аргументов называются ***argument matcher*** (**сопоставители аргументов**). Одновременно использовать матчеры и конкретные значения нельзя.

Метод ***.thenReturn()*** позволяет определить, что именно должен делать мок.

Метод ***.thenAnswer()*** позволяет использовать поступившие данные для формирования ответа. Объект *invocationOnMock* содержит в себе аргументы, с которыми был вызван метод *getAuthorDescription*.

Метод ***invocationOnMock.getArgument()*** возвращает один конкретный аргумент вызова. Первым параметром указывается индекс аргумента, начиная с нуля, а вторым – его тип данных.

*Mockito*

*.when(mockAuthorService.getAuthorDescription(anyInt()))*

*.thenAnswer(invocationOnMock -> {*

*int authorId = invocationOnMock.getArgument(0, Integer.class);*

*if (authorId % 2 == 0) {*

*return "великий русский писатель"; // чётные id у русских авторов*

*} else {*

*return "великий английский писатель"; } }); // нечётные id у английских авторов*

Метод ***.thenThrow()*** позволяет выбросить определенное исключение. Само исключение передается параметром.

*Mockito*

*.when(mockAuthorService.getAuthorDescription(Mockito.anyInt()))*

*.thenThrow(new DataNotAvailableException("Ошибка при доступе к базе"));*

Метод ***Mockito.verify()*** позволяет проверить содержимое мока. Например, проверим, что у мока *orderDao* метод *saveOrder()* был вызван ровно один раз с указанными аргументами.

*Mockito.verify(orderDao, Mockito.times(1))*

*.saveOrder(2, "адрес", 5, 1, deliveryDate);*

Метод ***Mockito.verifyNoInteractions()*** позволяет проверить, что во время работы теста не происходило никаких вызовов к мокам из списка.

Метод ***Mockito.verifyNoMoreInteractions()*** позволяет проверить, что во время работы теста не происходило никаких вызовов к мокам, не входящим в список.

*Mockito.verifyNoMoreInteractions(customerService, bookService, orderDao);*

Условие ***Mockito.times(int)*** задает ожидаемое количество вызовов. Также можно использовать методы ***atLeast()*** или ***atMost()***, задающие минимальное или максимальное допустимое количество вызовов.

Условие ***Mockito.never()*** проверяет, что вызовов с такими аргументами не было.

Подробнее о методах *Mockito* можно почитать [тут](https://site.mockito.org/).

Аннотация ***@Mock*** позволяет создать мок на весь тестовый класс. При этом тестовый класс нужно пометить аннотацией ***@ExtendWith(MockitoExtension.class)*** для корректной генерации мока.

*@ExtendWith(AuthorService.class)*

*public class BookServiceTestByAnnotations {*

*@Mock*

*AuthorService mockAuthorService;*

*@Test*

*void testCreateBookDescriptionWithMockito() {*

*BookService bookService = new BookService();*

*bookService.setAuthorService(mockAuthorService);*

*Mockito*

*.when(mockAuthorService.getAuthorDescription(Mockito.anyInt()))*

*.thenReturn("великий русский писатель");*

*String bookDescription = bookService.createBookDescription("Война и мир", 1898, 5, "Л.Н.Толстой");*

*Assertions.assertEquals("Война и мир, 1898 автор Л.Н.Толстой, великий русский писатель",*

*bookDescription); } }*

Вспомогательный класс ***ReflectionTestUtils*** позволяет получать доступ к приватным полям и методам в тестовых целях. Он помогает обойти *dependency injection* через аннотации *@Autowired*.

*@Service*

*public class OrderService {*

*@Autowired*

*CustomerService customerService;*

*@Autowired*

*BookServiceInternal bookService;*

*@Autowired*

*OrderDao orderDao; }*

*@Test*

*void testSaveOrder() {*

*OrderService orderService = new OrderService();*

*CustomerService customerService = Mockito.mock(CustomerService.class);*

*BookService bookService = Mockito.mock(BookService.class);*

*OrderDao orderDao = Mockito.mock(OrderDao.class);*

*ReflectionTestUtils.setField(orderService, "customerService", customerService);*

*ReflectionTestUtils.setField(orderService, "bookService", bookService);*

*ReflectionTestUtils.setField(orderService, "orderDao", orderDao);*

*// проверки }*

Аннотация ***@SpringJUnitConfig( { PersistenceConfig.class, MockitoExtension.class} )*** уже включает в себя аннотацию ***@ExtendWith*** и создает *Spring*-контекст. Обычно используется в чистом *Spring Framework*.

Автоматическое внедрение зависимостей по умолчанию не работает в тестах. Аннотация ***@RequiredArgsConstructor(onConstructor\_ = @Autowired)*** указывает *Lombok* на необходимость генерирования конструктора для ручного внедрения зависимостей.

Аннотация ***@TestPropertySource*** позволяет задать настройки (или файл с настройками) для подключения к тестовой БД. Имеет более высокий приоритет, чем *application.properties*.

*@Transactional*

*@RequiredArgsConstructor(onConstructor\_ = @Autowired)*

*@TestPropertySource(properties = { "db.name=test"}) // переопределение только имени БД*

*@SpringJUnitConfig( { PersistenceConfig.class, UserServiceImpl.class})*

*class UserServiceImplTest { … }*

Задать настройки тестовой БД можно также в *application.properties* с помощью профилей жизненного цикла *Maven*. Например,

spring.config.activate.on-profile=ci,test  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.url=jdbc:h2:mem:shareit  
spring.datasource.username=test  
spring.datasource.password=test

Для тестирования, кроме *JUnit*, можно использовать библиотеку ***Harmcrest***. Подробнее [тут](https://hamcrest.org/JavaHamcrest/).

Аннотация ***@SpringBootTest*** уже включает в себя аннотации ***@SpringJUnitConfig*** и ***@TestPropertySource***.

*@Transactional*

*@SpringBootTest(*

*properties = "db.name=test",*

*webEnvironment = SpringBootTest.WebEnvironment.NONE)*

*@RequiredArgsConstructor(onConstructor\_ = @Autowired)*

*class UserServiceImplTest { … }*

Аннотация ***@Transactional*** перед классом означает, что каждый тест будет запускаться в отдельной транзакции и откатываться в его конце. Таким образом в БД не будут попадать тестовые данные.

Аннотация ***@Rollback(false)*** отключает откат транзакции по окончанию теста. В этом случае все изменения сохраняются. Может использоваться как на уровне класса, так и на уровне теста.