**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; }*

**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, 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")*

***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**.

Каждой записи добавляется специальный признак – **первичный ключ** (***primary key***). Он обеспечивает уникальность записей и может быть представлен набором любых символов.

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



Операторы обращения к БД:

* ***SELECT*** – описывает что выгружать из базы.
* ***FROM*** – описывает откуда именно выгружать данные.
* ***LIMIT*** – указывает количество первых записей, которые нужно выгрузить. Ставят в конце запроса.

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

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.

Типы данных в PostgreSQL: