<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. Она объединяет в себе три других аннотации - ***@SpringBootConfiguration*** (аналог ***@Configuration***, объявляющей класс как класс конфигурации), ***@EnableAutoConfiguration*** (включает автоконфигурацию *Spring Boot*) и ***@ComponentScan*** (включает сканирование компонентов для создания бинов).

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; }*

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*. Если классы с аннотациями для создания бинов находятся в других пакетах, то для изменения пути их поиска можно использовать параметр ***scanBasePackages***.

*@SpringBootApplication(scanBasePackages = {"ru.practicum.stats\_client", "ru.practicum.main\_service"})*

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. Для полей класса можно использовать как эту аннотацию, так и ***@JsonFormat*** (потребуется конструктор без параметров). Эти аннотации используют указанный паттерн при сериализации/десериализации времени в/из строку. 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); }*

*@JsonFormat(pattern = "yyyy-MM-dd", shape = JsonFormat.Shape.STRING)*

*private LocalDate 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).

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*.

***Swagger*** – это фреймворк для спецификации *RESTful API*. Он позволяет интерактивно просматривать документацию, а также отправлять запросы с помощью инструмента ***Swagger UI***. Также по спецификации *API Swagger* может сгенерировать клиент или сервер. Подробнее можно почитать [тут](https://swagger.io/). Ссылка на [онлайн редактор](https://editor-next.swagger.io/).