

Введение в Spring Framework

Что это такое

- Универсальный фреймворк для разработки приложений на Java (не только «кروавый энтерпрайз!»).
- Открытый исходный код, первая версия вышла в 2003 г.
- Реализует паттерн IoC и механизмы CDI.
- Активно использует инфраструктурные решения Java / Jakarta EE.
- «Фреймворк фреймворков».

«Идейные» отличия от Java EE

- «Базовая» концепция Java EE – разделение обязанностей между контейнером и компонентом; «базовая» концепция Spring – IoC / CDI.
- Контейнер в Java EE включает в себя приложение; приложение в Spring включает в себя контейнер.
- Java EE – спецификация; Spring – фреймворк.

0. «Фреймворк фреймворков»

Многообразие проектов внутри

- Spring Core;
- Spring Web MVC;
- Spring WebFlux;
- Spring Data (JDBC, REST, JPA);
- Spring Security;
- Spring Cloud;
- Spring Boot;
- МНОГО ИХ.

«Джентельменский» наборчик

- Spring Boot — управление совместимостью библиотек и фреймворков + автоконфигурация;
- Spring Web MVC — REST API, реже + фронт на основе шаблонизаторов, совсем реже + фронт на JSP\JSF;
- Spring Core — само «связующее ядро» + бины для бизнес-логики;
- Spring Data JPA + Hibernate — доступ к данным;
- Spring Security — безопасность.

1. Spring Core

ApplicationContext (1)

- «Сердце» любого Spring-based приложения;
- Обобщенный интерфейс со множеством реализаций:
 - GenericApplicationContext — **программная** конфигурация бинов;
 - ClassPathXmlApplicationContext — конфигурация бинов при помощи **xml**;
 - AnnotationConfigApplicationContext — конфигурация бинов при помощи **аннотаций**;

Пример (1)

```
public static void main(String[] args) {  
    ApplicationContext ctx = new  
    AnnotationConfigApplicationContext(AppConfig.class);  
    MyBean bean = ctx.getBean(MyBean.class);  
    // теперь можно использовать bean  
}
```

ApplicationContext (2)

ApplicationContext наследуется от BeanFactory — «фабрика с рюшечками» ;

Отличия от BeanFactory:

- поддержка @Component, @Configuration и т. д.;
- поддержка методов ЖЦ @PostConstruct и @PreDestroy;
- Spring's resources, events и т. д.

Пример (2)

AppConfig.java:

```
@Configuration // указывает, что это конфигурация
public class AppConfig {
    @Bean // метод-провайдер бина
    public MyBean provideMyBean() {
        return new MyBean("Сессия близко");
    }
}
```

MyBean.java:

```
public class MyBean {
    private final String messageToStudents;
    public MyBean(String message) {
        this.messageToStudents = message;
    }
    // какие-то методы
}
```

ApplicationContext (3)

Два способа конфигурации:

- **xml** — устаревший вариант;
- **annotations** — при помощи «сканирования»;

При сканировании выполняется поиск бинов, помеченных `@Component` и `@Configuration`.

В `@Configuration` классах можно объявить методы-провайдеры бинов, пометив их аннотацией `@Bean`.

`@ComponentScan` — для указания пакетов, в которых нужно выполнить сканирование.

Пример (3.1)

AppConfig.java:

```
@Configuration
```

```
@ComponentScan("org.itmo.web.beans") // где искать
```

```
public class AppConfig {}
```

org.itmo.web.beans.MyBean.java:

```
@Component // что искать
```

```
public class MyBean {
```

```
    public MyBean() {
```

```
        // обязательно конструктор без параметров
```

```
    }
```

```
}
```

Пример (3.2)

```
org.itmo.web.beans.MyBean.java:
@Component
public class MyBean {
    private final String messageToStudents;
    // пустой конструктор можно опустить

    @PostConstruct // без параметров, возвращаем void
    public void fillMessage() {
        this.messageToStudents = "Рубежка близко";
    }
}
```

Области видимости бинов

- Singleton — по умолчанию;
- Prototype — новый объект на каждый запрос бина у контекста;

Только для web-контекста:

- Request;
- Session;
- Application;
- WebSocket.

Области видимости бинов

- Singleton — по умолчанию;
- Prototype — новый объект на каждый запрос бина у контекста;

Только для web-контекста:

- Request;
- Session;
- Application;
- WebSocket.

Пример (4)

```
@Component // для prototype - аналогично
// @Scope("singleton") - @SingletonScope не существует!
@Scope(value = WebApplicationContext.SCOPE_SINGLETON)
public class MyBean {}

@Configuration
public class AppConfig {
    @Bean
    @RequestScope // @SessionScope, @ApplicationScope
    public AnotherBean provideAnotherBean() { // ... }
}
```

Dependency injection

- когда нужен бин — он запрашивается у `ApplicationContext`;
- до Spring Framework 4.3 нужно было явно указывать аннотацию `@Autowired`;
- «по умолчанию» — внедрение через конструктор;
- используя `@Autowired` — можно внедрять через setter;
- также поддерживаются аннотации из JSR-330 — `@Inject`, `@Named`.

Пример (5)

@Component

```
public class MyBean {  
    private final AnotherBean anotherBean;  
    private final OneMoreBean oneMoreBean;  
  
    public MyBean(AnotherBean anotherBean) {  
        this.anotherBean = anotherBean;  
    }  
  
    @Autowired  
    public void setOneMoreBean(OneMoreBean oneMoreBean) {  
        this.oneMoreBean = oneMoreBean;  
    }  
}
```

Разные стратегии поиска бинов (1)

- Spring ищет бины по требуемому типу;
- можно использовать `@Primary` и `@Qualifier` в случае, если бинов одного типа несколько;
- каждому бину присваивается имя:
 - по умолчанию — имя класса с маленькой буквы;
 - можно изменить при помощи `@Component("name")` или `@Qualifier("name")`;
- можно создавать свои аннотации-спецификаторы на основе `@Qualifier`.

Пример (6.1)

```
public interface StudentExpellStrategy {}
```

```
@Component("randomExpell")
```

```
public class RandomStudentExpellStrategy implements StudentExpellStrategy{}
```

```
@Component("progressExpell")
```

```
public class ProgressStudentExpellStrategy implements StudentExpellStrategy{}
```

Пример (6.2)

```
@Component
public class MyBean {
    private final StudentExpellStrategy strategy;

    public MyBean(
        @Qualifier("progressExpell") StudentExpellStrategy strategy
    ) {
        this.strategy = strategy;
    }
}
```

Пример (6.3)

```
public interface StudentExpellStrategy {}

public class RandomStudentExpellStrategy implements StudentExpellStrategy{}

public class ProgressStudentExpellStrategy implements StudentExpellStrategy{}

@Configuration
public class ExpellConfiguration {
    @Bean
    public ProgressStudentExpellStrategy provideA() { // ... }

    @Bean
    @Primary
    public RandomStudentExpellStrategy provideB() { // ... }
}
```

Разные стратегии поиска бинов (2)

В случае generic-типов Spring использует информацию о типе параметризации как спецификатор.

```
@Component
public class BeanA implements List<String> {}
```

```
@Component
public class BeanB implements List<Long> {}
```

```
@Component
public class MyBean {
    private final List<String> list;

    public MyBean(List<String> list) {
        this.list = list;
    }
}
```


Environment (1)

- Spring Core предоставляет абстракцию Environment;
- через Environment можно получить доступ:
 - к переменным окружения;
 - к JVM переменным;
- С абстракцией Environment тесно связано понятие «профиля запуска приложения»;
- можно наполнить ApplicationContext бинами в зависимости от профиля запуска.

Пример (7.1)

```
public interface MailService {}

/**
 * Реализует отправку писем через почтовый сервер GMAIL.
 */
@Component
@Profile("prod")
public class GmailMailService implements MailService {}

/**
 * Заглушка для локальных тестов на машине разработчика
 */
@Component
@Profile("dev")
public class StubMailService implements MailService {}
```

Пример (7.1)

```
AnnotationConfigApplicationContext ctx = new
    AnnotationConfigApplicationContext(AppConfig.class);
ctx.getEnvironment().setActiveProfiles("dev"); // выбираем профиль запуска
ctx.refresh(); // нужно выполнить повторное сканирование

@Configuration
@ComponentScan("org.itmo.web.mail")
public class AppConfig {}
```

Environment (2)

- можно добавлять дополнительные источники параметров конфигурации - `@PropertySource`;
- источники параметров просматриваются в следующем порядке:
 - пользовательские источники;
 - `ServletConfig` (только для web-контекста);
 - `ServletContext` (`<context-param>` в `web.xml`);
 - JNDI;
 - параметры запуска JVM;
 - переменные окружения.

Пример (7.3)

```
@Configuration
@PropertySource("classpath:/application.yml")
public class AppConfig {
    private final Environment env;
    private final String someProp;

    // при помощи @Value можно инжектировать параметры конфигурации
    public AppConfig(Environment env, @Value("${some.prop}") String prop) {
        this.env = env; this.someProp = prop;
    }

    @Bean
    public MyBean provideMyBean() {
        // при помощи getProperty() у Environment можно получить значения
        return new MyBean(env.getProperty("message.to.students"));
    }
}
```

← Вот там был необходимый минимум

Однако в Spring Core много и других
«интересностей»

Для искушенных зрителей:

<https://docs.spring.io/spring-framework/reference/core.html>

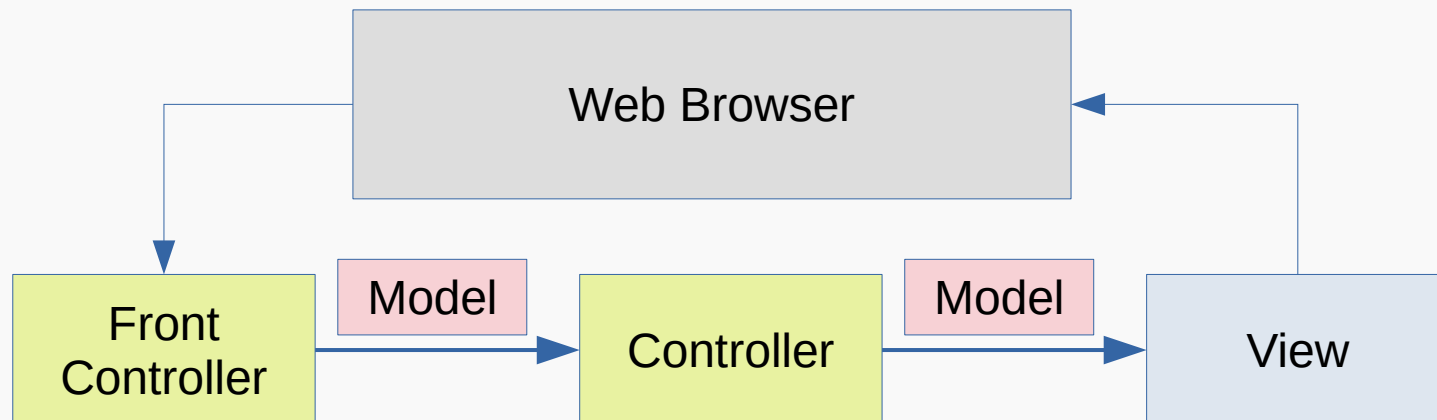
2. Spring Web MVC

Общие моменты

- Spring Web MVC – “базовый” фреймворк в составе Spring для разработки веб-приложений.
- Основан на паттерне MVC (внезапно!)
- Back-end; универсальный, удобен для разработки REST API.
- На клиентской стороне интегрируется с популярными JS-фреймворками.
- Удобно интегрируется с Thymeleaf.

Из чего состоит приложение

- **Model** – инкапсулирует данные приложения (состоят из POJO или бинов).
- **View** – отвечает за отображение данных модели.
- **Controller** – обрабатывает запрос пользователя, создаёт соответствующую модель и передаёт её для отображения в представление.



Модель

- Хранит данные, необходимые для формирования представления.
- Сами по себе эти данные – обычные POJO.
- В общем случае, реализует интерфейс `org.springframework.ui.Model`.
- Есть «упрощённая» реализация, представляющая из себя Map – `org.springframework.ui.ModelMap`.

Model:

```
@GetMapping("/showViewPage")
public String passParametersWithModel(Model model) {
    Map<String, String> map = new HashMap<>();
    map.put("spring", "mvc");
    model.addAttribute("message", "Hello, World!");
    model.mergeAttributes(map);
    return "viewPage";
}
```

ModelMap:

```
@GetMapping("/printViewPage")
public String passParametersWithModelMap(ModelMap map) {
    map.addAttribute("welcomeMessage", "welcome");
    map.addAttribute("message", "Hello, World!");
    return "viewPage";
}
```

Контроллер

- Класс, который связывает модель с представлением, управляет состоянием модели.
- Помечается аннотацией `@Controller`.
- Класс или его методы могут быть помечены аннотациями, «привязывающими» их к определённым методам HTTP или URL.

Пример контроллера

@Controller

```
public class HelloController {  
    @RequestMapping(value = "/hello",  
                    method = RequestMethod.GET)  
    public String printHello(ModelMap  
                                model) {  
        model.addAttribute("message",  
                            "Hello Spring MVC Framework!");  
        return "hello";  
    }  
}
```

Представление

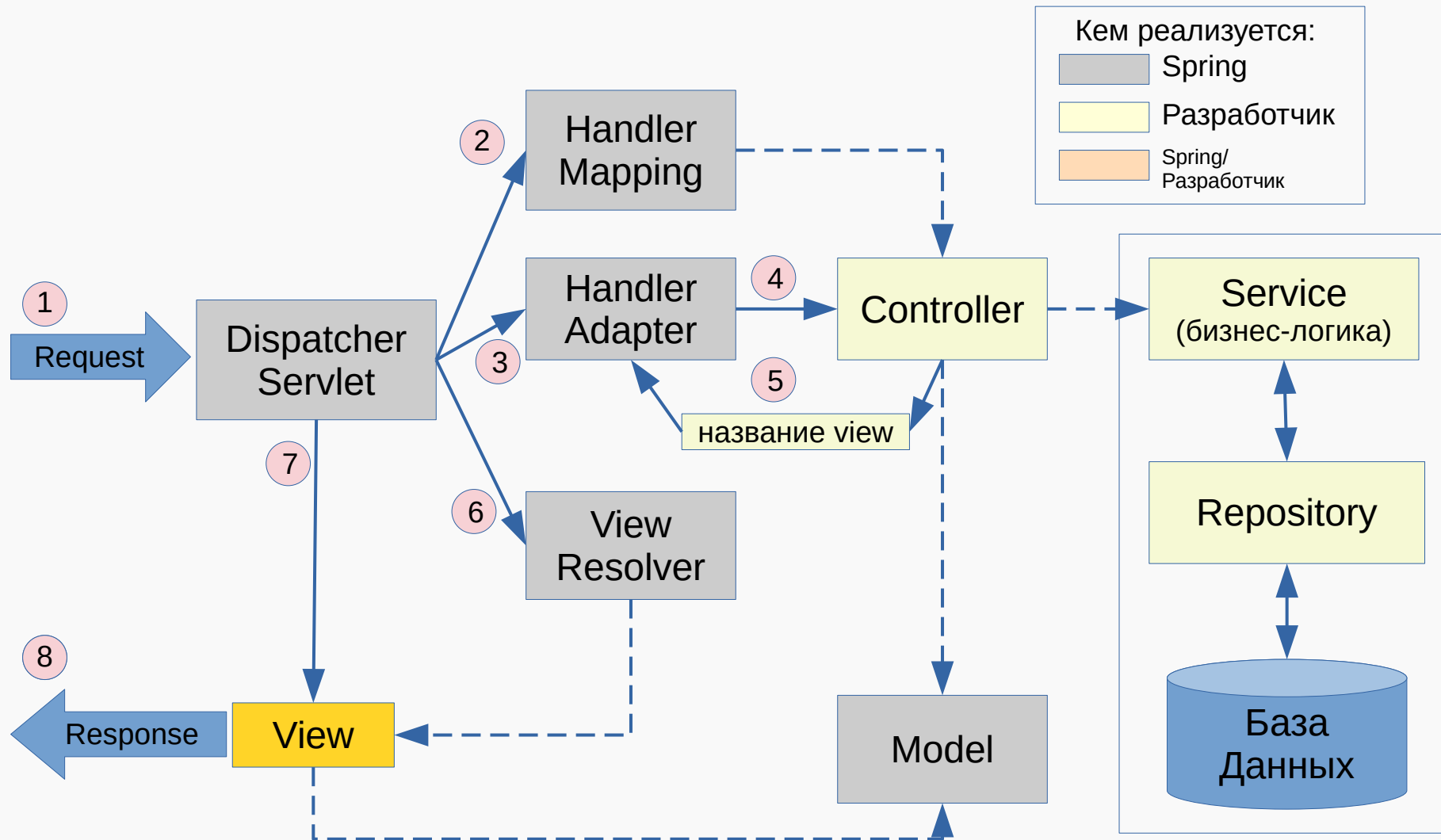
- Фреймворк не специфицирует жёстко технологию, на которой должно быть построено представление.
- Вариант «по-умолчанию» – JSP.
- Можно использовать Thymeleaf, FreeMarker, Velocity etc.
- Можно реализовать представление вне контекста Spring – целиком на JS.

Пример представления

На JSP:

```
<html>
  <head>
    <title>Hello Spring MVC</title>
  </head>
  <body>
    <h2>${message}</h2>
  </body>
</html>
```

Обработка запроса



Dispatcher Servlet

- Обработывает все запросы и формирует ответы на них.
- Связывает между собой все элементы архитектуры Spring MVC.
- Обычный сервлет – конфигурируется в `web.xml`.

Конфигурация web.xml

```
<web-app>
<servlet>
  <servlet-name>HelloWeb</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>HelloWeb</servlet-name>
  <url-pattern>*.jsp</url-pattern>
</servlet-mapping>
</web-app>
```

Handler Mapping

- Механизм, позволяющий распределять запросы по различным обработчикам.
- Помимо «основного» Handler'а, в обработке запроса могут участвовать один или несколько «перехватчиков» (реализаций интерфейса `HandlerInterceptor`).
- Механизм в общем похож на сервлеты и фильтры.
- «Из коробки» программисту доступно несколько реализаций `Handler Mapping`.

Конфигурация Handler Mapping

На примере BeanNameUrlHandlerMapping:

```
<beans>
  <bean id="handlerMapping"
        class="o.s.w.s.h.BeanNameUrlHandlerMapping"/>

  <bean name="/editaccount.form"
        class="o.s.w.s.m.SimpleFormController">
    <property name="formView" value="account"/>
    <property name="successView"
              value="account-created"/>
    <property name="commandName" value="account"/>
    <property name="commandClass"
              value="samples.Account"/>
  </bean>
</beans>
```

ViewResolver

- Представление в Spring Web MVC может быть построено на разных технологиях.
- С каждым представлением сопоставляется его символическое имя.
- Преобразованием символических имён в ссылки на конкретные представления занимается специальный класс, реализующий интерфейс `org.springframework.web.servlet.ViewResolver`.
- Существует много реализаций ViewResolver для разных технологий построения представления.
- В одном приложении можно использовать несколько ViewResolver'ов.

Конфигурация ViewResolver'a

Пример для UrlBasedViewResolver:

```
<bean id="viewResolver"
      class="o.s.w.s.v.UrlBasedViewResolver">
    <property name="viewClass"
              value="o.s.w.s.v.JstlView"/>
    <property name="prefix"
              value="/WEB-INF/jsp/" />
    <property name="suffix"
              value=".jsp" />
</bean>
```

Объединение ViewResolver'ов в последовательность

```
<bean id="jspViewResolver"  
      class="o.s.w.s.v.InternalResourceViewResolver">  
  <property name="viewClass"  
            value="o.s.w.s.v.JstlView"/>  
  <property name="prefix" value="/WEB-INF/jsp/" />  
  <property name="suffix" value=".jsp" />  
</bean>
```

```
<bean id="excelViewResolver"  
      class="o.s.w.s.v.XmlViewResolver">  
  <property name="order" value="1" />  
  <property name="location" value="/WEB-INF/views.xml" />  
</bean>
```

```
<!-- in views.xml →  
<beans>  
  <bean name="report"  
        class="o.s.e.ReportExcelView"/>  
</beans>
```

Пример конфигурации Spring Web MVC (0)

- Далее приведен пример конфигурации простого back-end приложения на базе Spring Web MVC;
- Используемая версия библиотек Spring — 5.3.31;
- Также есть интеграция с Thymeleaf;
- Пример разворачивается на Tomcat 9.0 (JDK11);
- Сборка при помощи Gradle 8.4;
- Пример проверен в декабре 2023.

Структура проекта:

```
nobot:
-- src/main
-- -- java
-- -- -- org/itmo/noboot
-- -- -- -- AppConfig.java
-- -- -- -- HelloWorldController.java
-- -- resources
-- -- -- templates/hello-world.html
-- -- webapp
-- -- -- WEB-INF/web.xml
-- build.gradle
-- setting.gradle
```

Список зависимостей:

build.gradle:

```
plugins {  
    id 'java'  
    id 'war'  
}  
  
group = 'org.itmo'  
version = '1.0-SNAPSHOT'  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation 'org.springframework:spring-core:5.3.31'  
    implementation 'org.springframework:spring-context:5.3.31'  
    implementation 'org.springframework:spring-web:5.3.31'  
    implementation 'org.springframework:spring-webmvc:5.3.31'  
    implementation 'org.thymeleaf:thymeleaf-spring5:3.1.2.RELEASE'  
}
```

Пример конфигурации Spring Web MVC (3)

```
package org.itmo.noboot;  
  
@Controller  
@RequestMapping("/hello-world")  
public class HelloWorldController {  
    @GetMapping  
    public String helloWorld() { return "hello-world"; }  
}
```

Пример конфигурации Spring Web MVC (4)

```
package org.itmo.noboot;

@Configuration
@ComponentScan(basePackages = "org.itmo.noboot")
public class AppConfig extends DelegatingWebMvcConfiguration {
    @Override
    protected void configureViewResolvers(ViewResolverRegistry registry) {
        var templateResolver = new ClassLoaderTemplateResolver();
        templateResolver.setPrefix("templates/");
        templateResolver.setSuffix(".html");
        templateResolver.setTemplateMode("HTML5");

        var springTemplateEngine = new SpringTemplateEngine();
        springTemplateEngine.setTemplateResolver(templateResolver);

        var thymeleafViewResolver = new ThymeleafViewResolver();
        thymeleafViewResolver.setTemplateEngine(springTemplateEngine);
        thymeleafViewResolver.setOrder(1);

        registry.viewResolver(thymeleafViewResolver);
    }
}
```

Пример конфигурации Spring Web MVC (5)

web.xml:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee">
  <servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
    <init-param>
      <param-name>contextClass</param-name>
      <param-value>
        org.springframework.web.context.support.AnnotationConfigWebApplicationContext
      </param-value>
    </init-param>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>org.itmo.noboot.AppConfig</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

  <!-- ПРОДОЛЖЕНИЕ ДАЛЕЕ -- >
```

Пример конфигурации Spring Web MVC (6)

web.xml:

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

<context-param>
  <param-name>contextClass</param-name>
  <param-value>
    org.springframework.web.context.support.AnnotationConfigWebApplicationContext
  </param-value>
</context-param>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>org.itmo.noboot.AppConfig</param-value>
</context-param>
</web-app>
```

3. Spring Boot

Проблема Spring

- Разные библиотеки\фреймворки разрабатываются разными членами сообщества;
- Между разными поделками — тесные взаимосвязи;
- Все эти библиотеки\фреймворки продолжают получать обновления каждый год;
- В результате имеем проблемы с совместимостью версий разных библиотек;
- Также у некоторых библиотек\фреймворков высокая сложность конфигурации.

Spring Boot

- Появился в 2014 году;
- Решает проблемы с совместимостью версий между библиотеками\фреймворками экосистемы Spring;
- Предоставляет «типовую» конфигурацию компонентов на основе автоконфигураций;
- Может быть «просто запущен» за счет использования embedded Java EE Application Server (обычно Tomcat).

Spring Boot starter

- `org.springframework.boot:spring-boot-starter-web`
- `org.springframework.boot:spring-boot-starter-security`
- `org.springframework.boot:spring-boot-starter-data-jpa`
- и множество других;

Есть плагины для Gradle и Maven для сборки.

Пример конфигурации Gradle

https://docs.gradle.org/current/samples/sample_building_spring_boot_web_applications.html

```
plugins {  
    id 'org.springframework.boot' version '2.7.8'  
    id 'java'  
}  
  
version = '1.0.2'  
group = 'org.gradle.samples'  
  
java {  
    sourceCompatibility = JavaVersion.VERSION_1_8  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation platform('org.springframework.boot:spring-boot-dependencies:2.7.8')  
    implementation 'org.springframework.boot:spring-boot-starter'  
}
```

Пример Spring Boot Application

MyApplication.java:

```
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

HelloWorldController.java

```
@RestController
@RequestMapping("/hello-world")
public class HelloWorldController {
    @GetMapping
    public String sayHello() {
        return "Hello, World!";
    }
}
```

Теперь нам не надо (1)

- Настраивать сканирование пакетов на наличие бинов вручную;
- **Практически** не нужно конфигурировать Spring Web MVC или любой другой фреймворк в составе Spring Framework;
- Не нужно поднимать Java EE Application Server самостоятельно и деплоить на него Spring Application;

Теперь нам не надо (2)

- Spring Boot подтягивает параметры конфигурации из файла `application.yml` в директории `resources` по умолчанию;
- Если используем профили запуска приложения, отличные от `default` — Spring Boot подтягивает параметры из файлов `application-{profile}.yml`;
- Можно использовать аннотацию `@ConfigurationProperties` вместо `@Value`.

Пример @ConfigurationProperties (1)

MyApplication.java:

```
@EnableConfigurationProperties
@ConfigurationPropertiesScan("org.itmo.web.properties")
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```



Пример @ConfigurationProperties (2)

MyProperties.java

```
@ConfigurationProperties(prefix = "itmo")
public class MyProperties {
    private final long studsToExpell;

    @ConstructorBinding
    public MyProperties(long studsToExpell) {
        This.studsToExpell = studsToExpell;
    }
}
```

application.yml

```
itmo:
  studs-to-expell: 1000
```


4. Spring Data JPA

Доступ к данным

- В «типовом» back-end приложении 20-50 таблиц — число на основе личных наблюдений;
- «Типовое» back-end приложение нужно для CRUD (Create Read Update Delete) + что-то еще;
- Писать SQL запросы долго — можно использовать ORM;
- Spring пошел еще дальше и придумал Spring Data JPA.

Пример репозитория (1)

@Repository // стереотип, по сути то же самое, что и **@Component**, но с семантическим значением

```
public interface StudentRepository extends JpaRepository<Student, Long> {}
```

```
// Тут JPA аннотации
```

```
@Entity
```

```
@Table(name = "students")
```

```
public class Student {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private long id;
```

```
    private String name;
```

```
    private String surname;
```

```
    private int age;
```

```
    // геттеры, сеттеры
```

```
}
```

Чаще всего компонент с аннотацией **@Repository** служит для реализации операций с БД, относящихся к указанной сущности (CRUD)

JPARepository

Modifier and Type	Method
void	<code>deleteAllByIdInBatch(Iterable <ID> ids)</code>
void	<code>deleteAllInBatch()</code>
void	<code>deleteAllInBatch(Iterable <T> entities)</code>
default void	<code>deleteInBatch(Iterable <T> entities)</code>
<S extends T> List <S>	<code>findAll(Example <S> example)</code>
<S extends T> List <S>	<code>findAll(Example <S> example, Sort sort)</code>
void	<code>flush()</code>
T	<code>findById(ID id)</code>
T	<code>getOne(ID id)</code>
T	<code>getReferenceById(ID id)</code>
<S extends T> List <S>	<code>saveAllAndFlush(Iterable <S> entities)</code>
<S extends T> S	<code>saveAndFlush(S entity)</code>

CRUDRepository

Modifier and Type	Method
long	<code>count()</code>
void	<code>delete(T entity)</code>
void	<code>deleteAll()</code>
void	<code>deleteAll(Iterable <? extends T> entities)</code>
void	<code>deleteAllById(Iterable <? extends ID> ids)</code>
void	<code>deleteById(ID id)</code>
boolean	<code>existsById(ID id)</code>
Iterable <T>	<code>findAll()</code>
Iterable <T>	<code>findAllById(Iterable <ID> ids)</code>
Optional <T>	<code>findById(ID id)</code>
<S extends T> S	<code>save(S entity)</code>
<S extends T> Iterable <S>	<code>saveAll(Iterable <S> entities)</code>

Пример репозитория (2)

@Service // стереотип, по сути то же самое, что и @Component, но с семантическим значением

```
public class StudentService {  
    private final StudentRepository studRepo;
```

Обычно компонент с аннотацией **@Service** служит для реализации бизнес-уровня приложения

```
    public StudentService(StudentRepository studRepo) {  
        this.studRepo = studRepo;  
    }
```

```
    public void createStudent(String name, String surname, int age) {  
        Student student = new Student();  
        student.setId(0L);  
        student.setName(name);  
        student.setSurname(surname);  
        student.setAge(age);  
        Student savedStudent = studRepo.save(student);  
        // id будет сгенерирован БД автоматически  
        System.out.println(savedStudent.id);  
    }  
}
```

Пример репозитория (3)

@Service // стереотип, по сути то же самое, что и @Component, но с семантическим значением

```
public class StudentService {  
    private final StudentRepository studRepo;  
  
    public StudentService(StudentRepository studRepo) {  
        this.studRepo = studRepo;  
    }  
  
    public Student getStudentById(long id) {  
        return studRepo.findById(id); // вернет null, если не найдено  
    }  
  
    public List<Student> getAllStudents() {  
        // вернет пустой список, если таблица пустая  
        return studRepo.findAll();  
    }  
}
```

Зачем нужен @Service

- @Service, @Repository, @Controller — «расширяют» @Component, дополняя семантическим значением (стереотипы);
- Технически различий нет — все бины @Service, @Repository, @Component — singleton;
- Сервис — бин, в котором хранится бизнес-логика приложения;
- Контроллер вызывает сервис; сервис вызывает репозиторий;
- Зачем так делать — вопрос философский!

- В интерфейсах-репозитория можно определить дополнительные методы для извлечения данных с использованием фильтров, определяемых предметной областью;
- Название сигнатуры превращается в JPQL запрос, который передается ORM-библиотеке под капотом;

Пример репозитория (4)

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {
    /** реализация будет «сгенерирована» автоматически
     * SELECT * FROM students WHERE name = :name;
     * ожидается 0 или 1 элемент в результате, если будет больше 1 – runtime
     exception */
    Student findByName(String name);
}

@Service
public class StudentService {
    private final StudentRepository studRepo;

    public StudentService(StudentRepository studRepo) {
        this.studRepo = studRepo;
    }

    public Student getStudentByName(String name) {
        return studRepo.findByName(name); // вернет null, если не найдено
    }
}
```

- Есть гораздо больше предикатов, чем простой WHERE;
- Поддержка DISTINCT, BETWEEN, LIKE, WHERE по полям вложенных сущностей и многое другое;
- В поисках полного перечня предикатов — welcome to official docs:

<https://docs.spring.io/spring-data/jpa/reference/repositories/query-methods-details.html>

application.yml:

spring:

datasource:

url: jdbc:postgresql://localhost:5432/my-database

driverClassName: org.postgresql.Driver // должен быть в classpath

username: my_user

password: my_password

build.gradle:

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

implementation 'org.postgresql:postgresql:42.7.1'

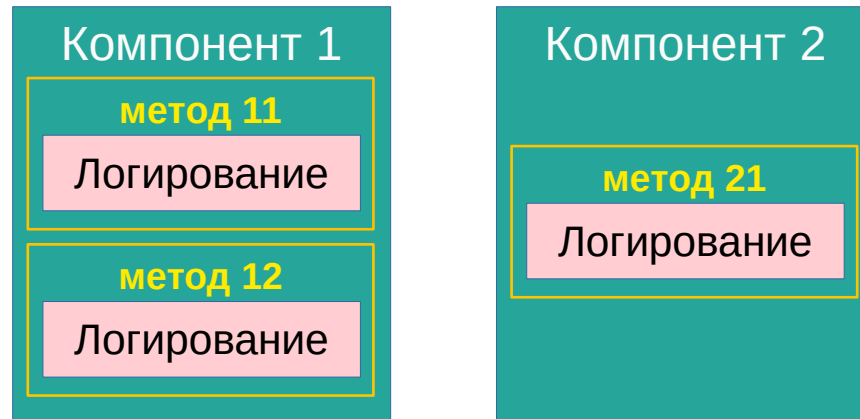
5. АОП

Аспектно-ориентированное программирование

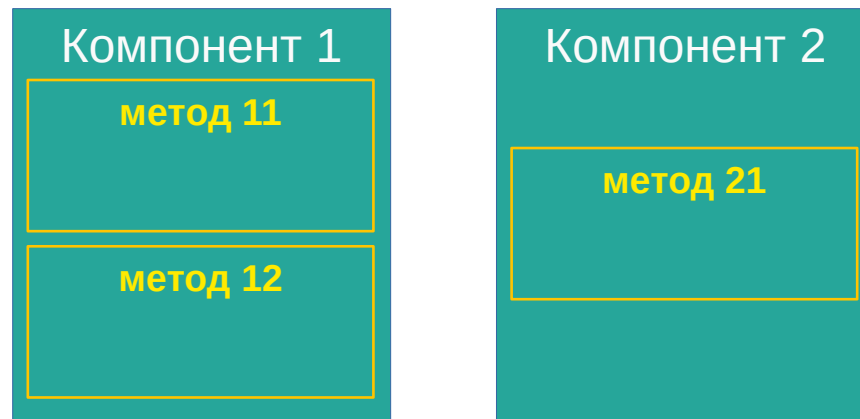
- АОП – парадигма программирования.
- Является развитием принципов процедурного программирования и ООП.
- Суть – внедрение в код “сквозной” функциональности – аспекта.
- “Каноничная” реализация в Java – AspectJ, она же используется в Spring.

Выделение “сквозной” функциональности

Было:

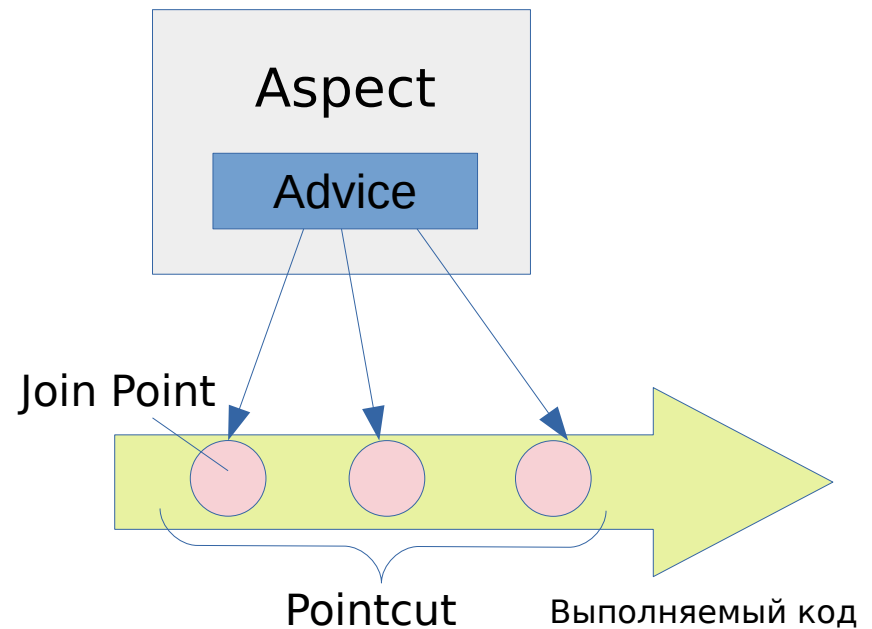


Стало:



АОП: основные понятия

- “Точка подключения” к коду (**join point**) — в ней добавляется “сквозная” функциональность.
- Именованный “срез” точек подключения (**pointcut**).
- Набор инструкций, выполняемых на pointcut’ах (**advice**).
- **Aspect** — модуль, в котором собраны описания advice’ов и pointcut’ов.



Пример: компонент с логированием

```
@Component
```

```
public class StudentComponent1 {
```

```
    public void method11() {
```

```
        System.out.println("method11 started...");
```

```
        //impl method11
```

```
    }
```

```
    public void method12() {
```

```
        System.out.println("method12 started...");
```

```
        //impl method12
```

```
    }
```

```
}
```

Вспомогательный код

Пример: компонент с логированием

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface StudentOperation { }
```

Аннотация StudentOperation:
поможет установить pointcut

Пример: убираем логирование

```
@Component  
public class StudentComponent1 {  
    @StudentOperation  
    public void method11() {  
        //impl method11  
    }  
    @StudentOperation  
    public void method12() {  
        //impl method12  
    }  
}
```

Аннотация StudentOperation:
отмечаем методы, которые
нужно логировать

Пример: убираем логирование

@Aspect

@Component

public class StudentAspect {

@Pointcut("@annotation(StudentOperation)")

public void callStudentComponentMethod() { }

@Before("callStudentComponentMethod()")

public void printCalledMethodName(JoinPoint joinPoint) {

String methodName =

joinPoint.getSignature().getMethod().getName();

System.out.println(methodName + " started...");

}

}

В аспекте создаем pointcut
для методов, которые
отмечены @StudentOperation

Пример: переносим логирование в аспект (1)

@Aspect

@Component

```
public class StudentAspect {
```

```
    @Pointcut("@annotation(StudentOperation)")
```

```
    public void callStudentComponentMethod() { }
```

```
    @Before("callStudentComponentMethod()")
```

```
    public void printCalledMethodName(JoinPoint joinPoint) {
```

```
        String methodName =
```

```
            joinPoint.getSignature().getMethod().getName();
```

```
        System.out.println(methodName + " started...");
```

```
    }
```

```
}
```

В аспекте
определяем
advice

Пример: pointcut без аннотаций

```
@Component  
public class StudentComponent1 {  
    public void method11() {  
        //impl method11  
    }  
    public void method12() {  
        //impl method12  
    }  
}
```

Пример: переносим логирование в аспект (2)

@Aspect

@Component

public class StudentAspect {

@Pointcut("execution(public * i.s.c.StudentComponent.*(..))")

public void callStudentComponentMethod() { }

@Before("callStudentComponentMethod()")

public void printCalledMethodName(JoinPoint joinPoint) {

String methodName =

joinPoint.getSignature().getMethod().getName();

System.out.println(methodName + " started...");

}

}

В аспекте создаем pointcut
для всех public
методов StudentComponent

Пример: переносим логирование в аспект (2)

@Aspect

@Component

public class StudentAspect {

С одним pointcut
может быть связано
несколько advice

@Pointcut("execution(public * i.s.c.StudentComponent.*(..))")

public void callStudentComponentMethod() { }

@Before("callStudentComponentMethod()")

public void printCalledMethodName(JoinPoint joinPoint) { ... }

@After("callStudentComponentMethod()")

public void printCalledMethodName(JoinPoint joinPoint) { ... }

}

Когда может выполняться advice?

@Aspect

@Component

```
public class StudentAspect {  
    @Pointcut("execution(public * i.s.c.S  
    public void callStudentComponentMetho
```

В данном примере advice
будет выполнен до pointcut.
Также возможно:
After, AfterReturning,
AfterThrowing, Around

```
    @Before("callStudentComponentMethod()")
```

```
    public void printCalledMethodName(JoinPoint joinPoint) {
```

```
        String methodName =
```

```
            joinPoint.getSignature().getMethod().getName();
```

```
        System.out.println(methodName + " started...");
```

```
    }
```

```
}
```