Технологии программирования

Spring Framework


Это еще что такое?

Что это такое?

Фреймворк - набор библиотек и правил по работе с ними.

Какие предоставляет преимущества?

Стандартизирует и структурирует проекты, предоставляет готовые решения для интеграций с различными технологиями.

То есть, буквально ускоряет и стандартизирует всю разработку, избавляя от необходимости в каждом проекте изобретать велосипед.

Помни, падаван, что не все фреймворк решает проблемы…

Плюсы использования фреймворков

- демонстрирует опыт и достижения ведущих разработчиков языка, на котором написан, «разрешая» применять эти достижения для своих задач;
- стандартизирует разработку, обеспечивая понимание архитектурных решений и кодовой базы многими разработчиками;
- предлагает готовые решения для:
  - безопасности приложения;
  - сериализации и десериализации;
  - интеграций, (СУБД, клиенты, серверы, брокеры сообщений и т.д.);
  - тестирования;
  - развертывания.

Минусы использования фреймворков

- предъявляет свои правила к написанию кода;
- ограничивает версии языка и библиотек;
- «раздувает» приложение;
- потенциально затрудняет поиск разработчиков.


spring®

Что это такое?

Spring – фреймворк для JVM языков. Задумывался и был реализован, как более удобная Java Enterprise Edition.

Преимущества Spring перед Java EE:

- spring совместим с Java EE (но не наоборот);
- разделен на проекты, (MVC, Data и пр.). Можно подключить только нужное;
- активно поддерживается сообществом.

Такое название имеет, потому что является олицетворением весны, новой жизни, пришедшей на смену зиме, (Java EE).

А зачем это нам нужно?

Исключая возможности, которые предоставляют «узкоспециализированные» проекты Spring, (взаимодействие с БД, WEB, security и т.д.), ядро фреймворка предоставляет нам: конфигурацию приложения, инверсию управления и внедрение зависимостей.

Инверсия управления

Инверсия управления (Inversion of control) — это идея разработки ПО, при которой управление объектами или частями программы передается в контейнер или платформу.

IoC дает несколько преимуществ:

- разделение выполнения задачи и ее реализации;
- упрощение переключения между различными реализациями;
- большая модульность программы;
- большее упрощение тестирования программы за счет изоляции компонента или имитации его зависимостей, а также разрешения компонентам взаимодействовать через контракты.

Инверсия управления в Spring

В Spring контейнером для IoC выступает интерфейс **Application Context**.

Он отвечает за создание, настройку и сборку объектов, которые в Spring называются Bean – компонентами.

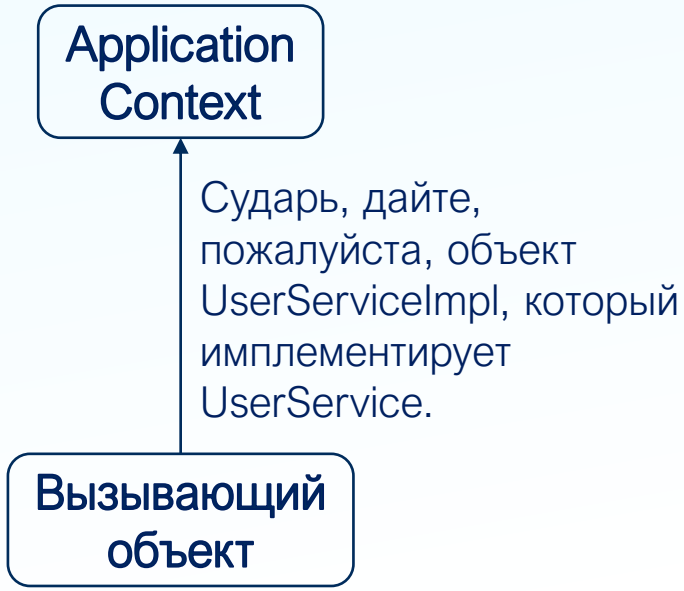
Также ApplicationContext отвечает за управление жизненными циклами бинов.

Spring Bean – это обычный объект Java, (POJO), в который закладывается дополнительная метаинформация, с помощью которой ApplicationContext распознает в нем сущность, которой он должен управлять.

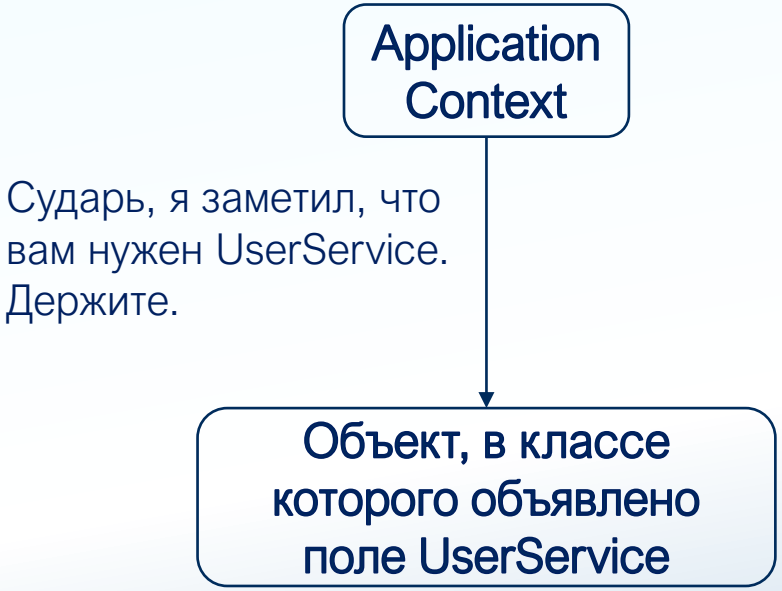
spring

Реализация инверсии управления в Spring

Поиск зависимостей – вызывающий объект запрашивает у контейнера экземпляр класса с определённым именем или типом.



Внедрение зависимостей – способ, в котором контейнер передает экземпляры объектов по их имени другим объектам.



spring

Реализация инверсии управления. Поиск зависимостей

```
class DependencyProvider {
    <T> T lookup(Class<T> type) {
        // code
    }
}

class TodoService implements Consumer<DependencyProvider> {
    TodoRepository todoRepository;

    @Override
    public void accept(DependencyProvider dependencyProvider) {
        this.todoRepository = dependencyProvider.lookup(TodoRepository.java);
    }
}
```

spring

Реализация инверсии управления. Внедрение зависимостей

Внедрение зависимостей в Spring реализуется несколькими способами:

– внедрение через конструктор;

– внедрение через set-метод;

– внедрение через свойства.

Внедрение зависимостей. Конструктор

```
class TodoService {
    TodoRepository todoRepsitory;
    TodoService(TodoRepository todoRepository) {
        this.todoRepository = todoRepository;
    }
}

class Application {
    public static void main(String[] args) {
        TodoRepository todoRepository = new TodoRepository();
        TodoService todoService = new TodoService(todoRepository);
    }
}
```

spring

Внедрение зависимостей. Конструктор. Spring

```
@Component
public class CarWithConstructor {
    private Engine engine;

    @Autowired
    public CarWithConstructor(Engine engine) {
        this.engine = engine;
    }

    public String toString() {
        return "car" + " with " + engine;
    }
}
```

spring

Внедрение зависимостей. Set-метод

```
class TodoService {
    TodoRepository todoRepository;
    void setTodoRepository(TodoRepository todoRepository) {
        this.todoRepository = todoRepository;
    }
}

class Application {
    public static void main(String[] args) {
        TodoRepository todoRepository = new TodoRepository();
        TodoService todoService = new TodoService();
        todoService.setTodoRepository(todoRepository);
    }
}
```

Внедрение зависимостей. Set-метод. Spring

```
@Component
public class CarWithSetter {

    private Engine engine;

    @Autowired
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    public String toString() {
        return "car" + " with " + engine;
    }
}
```

Внедрение зависимостей. Свойства

```
class TodoService {
    TodoRepository todoRepository;
}

class Application {
    public static void main(String[] args) {
        TodoService todoService = new TodoService();
        todoService.todoRepository = new TodoRepository();
    }
}
```

spring

Внедрение зависимостей. Свойства. Spring

```
@Component
public class CarWithSetter{
    @Autowired
    private Engine engine;



    public Engine getEngine() {
        return engine;
    }
}
```

Почему не магия. Этапы запуска Spring

Этап 1. Парсинг конфигурации и создание **BeanDefinition** всех бинов, которые нам понадобятся.

Этап 2. Настройка созданных **BeanDefinition**.

Этап 3. Создание кастомных **FactoryBean**.

Этап 4. **BeanFactory** создает экземпляры бинов, при необходимости делегируя создание бина **FactoryBean**.

Этап 5. Настройка созданных бинов.

Этап 1. Парсинг конфигурации и создание BeanDefiniton

У Spring есть 4 способа конфигурации:

- XML конфигурация,
  Конструктор: ClassPathXmlApplicationContext(“context.xml”).
- Groovy конфигурация,
  Конструктор: GenericGroovyApplicationContext(“context.groovy”).
- Конфигурация через аннотации с указанием пакета для сканирования,
  Конструктор: AnnotationConfigApplicationContext(“package.name”).
- Java Config,
  Конструктор: AnnotationConfigApplicationContext(JavaConfig.class).

XML конфигурация

```
<beans>
   <bean class="com.inwhite.spring.compare.CoolDaoImpl" id="coolDao"/>

   <bean id ="coolService” class="com.inwhite.spring.compare.CoolServiceImpl”
                              init-method="init"
                              destroy-method="closeResources”
                              scope="prototype">
      <property name="dao" ref="coolDao"/>
   </bean>
   …
</beans>
```

XML конфигурация

Для XML конфигурации используется класс — **XmlBeanDefinitionReader**, который реализует интерфейс **BeanDefinitionReader**.

1. **XmlBeanDefinitionReader** принимает на вход XML файл конфигурации.
2. Далее обрабатывается каждый элемент документа и если он является бином, то создается **BeanDefinition** на основе заполненных данных.
3. Каждый **BeanDefinition** вместе со своим именем помещается в **BeanDefinitionMap**.

Groovy конфигурация

```
beans {
        coolDao(CoolDaoImpl)

        coolService(CoolService){bean->
                bean.scope = 'prototype’
                bean.initMethod = 'init’
                bean.destroyMethod = 'closeResources’
        }
}
```

Groovy конфигурация

Для Groovy конфигурации все аналогично конфигурации через XML, только тут не XmlBeanDefinitionReader, а GroovyBeanDefinitionReader.

GroovyBeanDefinitionReader *прекрасно разберется и с XML файлом конфигурации.*

Конфигурация через аннотации

```
@Service
@Scope(BeanDefinition.SCOPE_PROTOTYPE)
public class CoolServiceImpl implements CoolService
{
        @Autowired
        private CoolDao dao;

        @PostConstruct
        public void init() {
                //init logic here
        }

        @PreDestroy
        public void closeResources() {
                //close resources here
        }

        @Override
        public void doWork() {
                dao.doCRUD();
        }
}
```

```
@Repository
public class CoolDaoImpl implements CoolDao {

        @Override
        public void doCRUD() {
                //some logic here
        }
}
```

spring

Java Config

```
@Configuration
public class JavaConfig {
        @Bean
        public CoolDao dao(){
                return new CoolDaoImpl();
        }

        @Bean(initMethod = "init", destroyMethod = "closeResources")
        @Scope(BeanDefinition.SCOPE_PROTOTYPE)
        public CoolService coolService(){
                CoolServiceImpl service = new CoolServiceImpl();
                service.setDao(dao());
                return service;
        }
}
```

Конфигурация через аннотации или Java Config

В обоих случаях используется класс **AnnotationConfigApplicationContext**.

```
new AnnotationConfigApplicationContext(“package.name”);
new AnnotationConfigApplicationContext(JavaConfig.class);
```

У этого контекста есть два поля:

```
private final ClassPathBeanDefinitionScanner scanner;
private final AnnotatedBeanDefinitionReader reader;
```

spring

Конфигурация через аннотации

1. Указанный пакет сканируется объектом **ClassPathBeanDefinitionScanner** на наличие классов помеченных аннотацией **@Component** или ее наследниками.
2. Найденные классы парсятся и для них создаются **BeanDefinition**.
3. Каждый BeanDefinition вместе со своим именем помещается в **BeanDefinitionMap**.

Чтобы сканирование было запущено, над классом, в котором находится метод main должен быть указан пакет для сканирования.

```
@ComponentScan({"package.name"})
```

Java Config

AnnotatedBeanDefinitionReader работает в несколько этапов.

1. Первый этап — это регистрация всех **@Configuration** для дальнейшего парсинга.
2. Второй этап — **ПАРСИНГ И СОЗДАНИЕ BeanDefinition**, (это регистрация специального BeanFactoryPostProcessor, а именно BeanDefinitionRegistryPostProcessor, который при помощи класса ConfigurationClassParser парсит JavaConfig и создает **BeanDefinition**).
3. Каждый **BeanDefinition** вместе со своим именем помещается в **BeanDefinitionMap**.

Сравнение

	XML	Аннотации	Java Config
Централизованная конфигурация	+	-	+
Отсутствие Spring в коде	+	-	+
Изменение конфигурации требует перекомпиляции	+	-	-
Типобезопасность	-	+	+
Полная поддержка IDE	-	-	+
Декларация бинов из сторонних бибилиотек	+	-	+
Возможность писать логику в конфигурации	-	-	+

BeanDefinition

Описание, как создавать и управлять конкретным бином.

Содержит:

- имя класса с указанием пакета;
- элементы поведенческой конфигурации бина, которые определяют, как бин должен вести себя в контейнере (scope, обратные вызовы жизненного цикла и т.д.);
- ссылки на другие bean-компоненты, которые необходимы для его работы. Эти ссылки также называются зависимостями;
- другие параметры конфигурации для установки во вновь созданном объекте, например, ограничение размера пула или количество соединений, используемых в бине, который управляет пулом соединений.

Итоги первого этапа

Spring обошел все конфигурацию, (даже, если использовалось несколько типов конфигурации) и создал **BeanDefinitionMap**, в которой лежат **BeanDefinition** для каждого, необходимого нам, бина.

Этап 2. Настройка созданных BeanDefinition

В Spring мы можем повлиять на то, какими будут наши бины еще до того, как они создадутся.

BeanFactoryPostProcessor

Благодаря интерфейсу **BeanFactoryPostProcessor** мы можем получить доступ к созданным **BeanDefinition** и можем их изменять.

```
public interface BeanFactoryPostProcessor {

void postProcessBeanFactory(
                ConfigurableListableBeanFactory beanFactory)
                        throws BeansException;

}
```

BeanFactoryPostProcessor

Метод **postProcessBeanFactory** принимает параметром **ConfigurableListableBeanFactory**.

Данная фабрика содержит много полезных методов, в том числе **getBeanDefinitionNames**, через который мы можем получить все BeanDefinitionNames.

А уже потом по конкретному имени получить *BeanDefinition* для дальнейшей обработки метаданных.

BeanFactoryPostProcessor

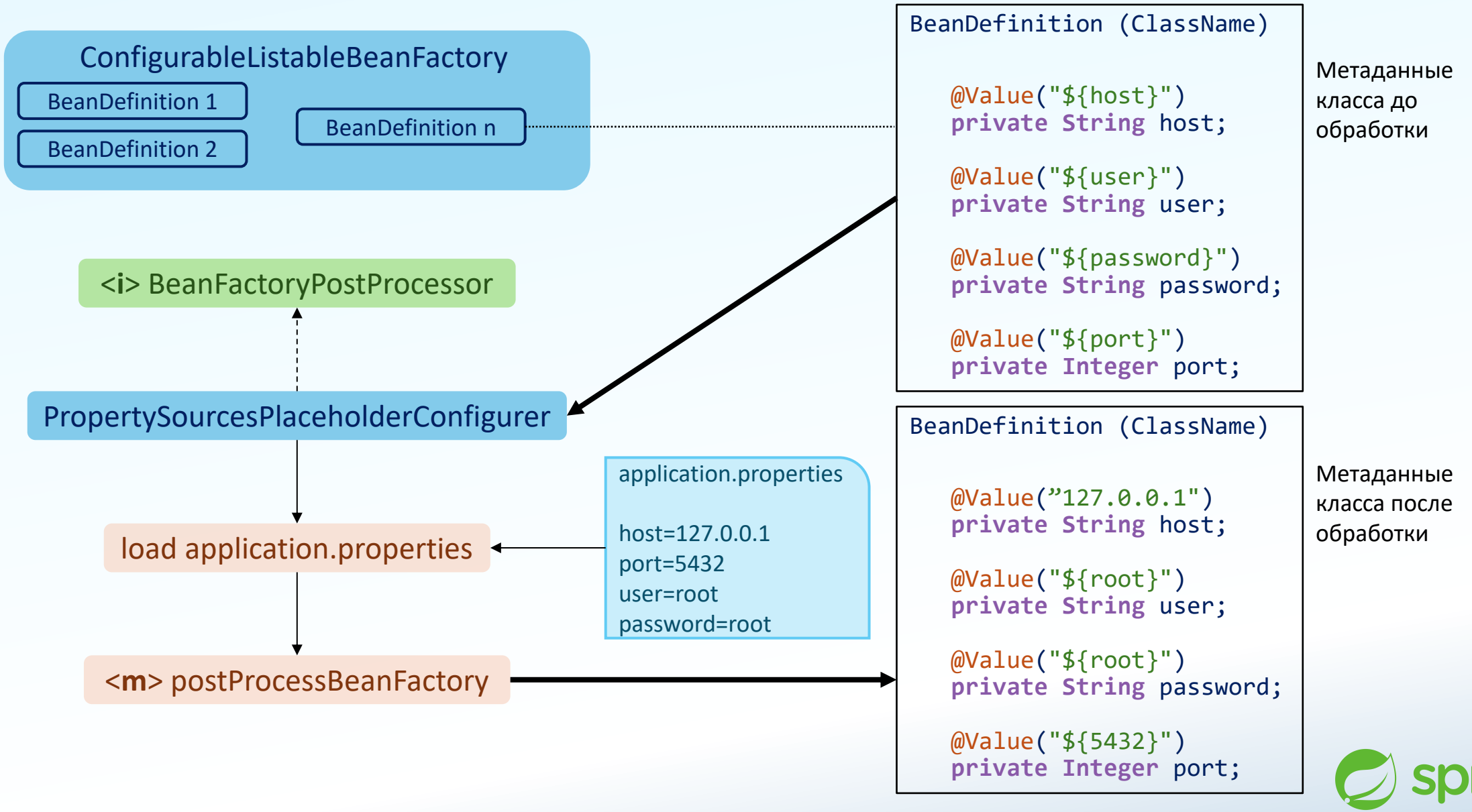
Реализация

PropertySourcesPlaceholderConfigurer – имплементация **BeanFactoryPostProcessor**, которая заменяет в месте назначения, (пример справа), ключи файла конфигурации на их значения.

```
@Component
public class ClassName {

        @Value("${host}")
        private String host;

        @Value("${user}")
        private String user;

        @Value("${password}")
        private String password;

        @Value("${port}")
        private Integer port;

}
```

spring

BeanFactoryPostProcessor



Итоги второго этапа

Spring обогатил и дополнил все **BeanDefinition**. И он готов создавать из этих описаний так необходимые нам бины.

Этап 3. Создание кастомных FactoryBean, (рудимент)

Когда-то Spring конфигурировался только через XML и разработчикам очень хотелось добавлять какое-либо поведение при создании бина.

Из этого запроса появился интерфейс **FactoryBean**.

FactoryBean

Для реализации необходимо передать класс бина, которым будет управлять фабрика в дженерик интерфейса:

```
public class ExampleFactory implements FactoryBean<Example>
```

А также реализовать 3 метода: getObject(), getobjectType(), isSingleton().

После этого необходимо зарегистрировать эту фабрику в XML конфигурации, как обычный бин.

Итоги третьего этапа

Разработчик указал Spring, какие бины нужно создавать не по умолчанию, а по определенным правилам, прописанным разработчиком.

Теперь Spring точно готов создавать бины.

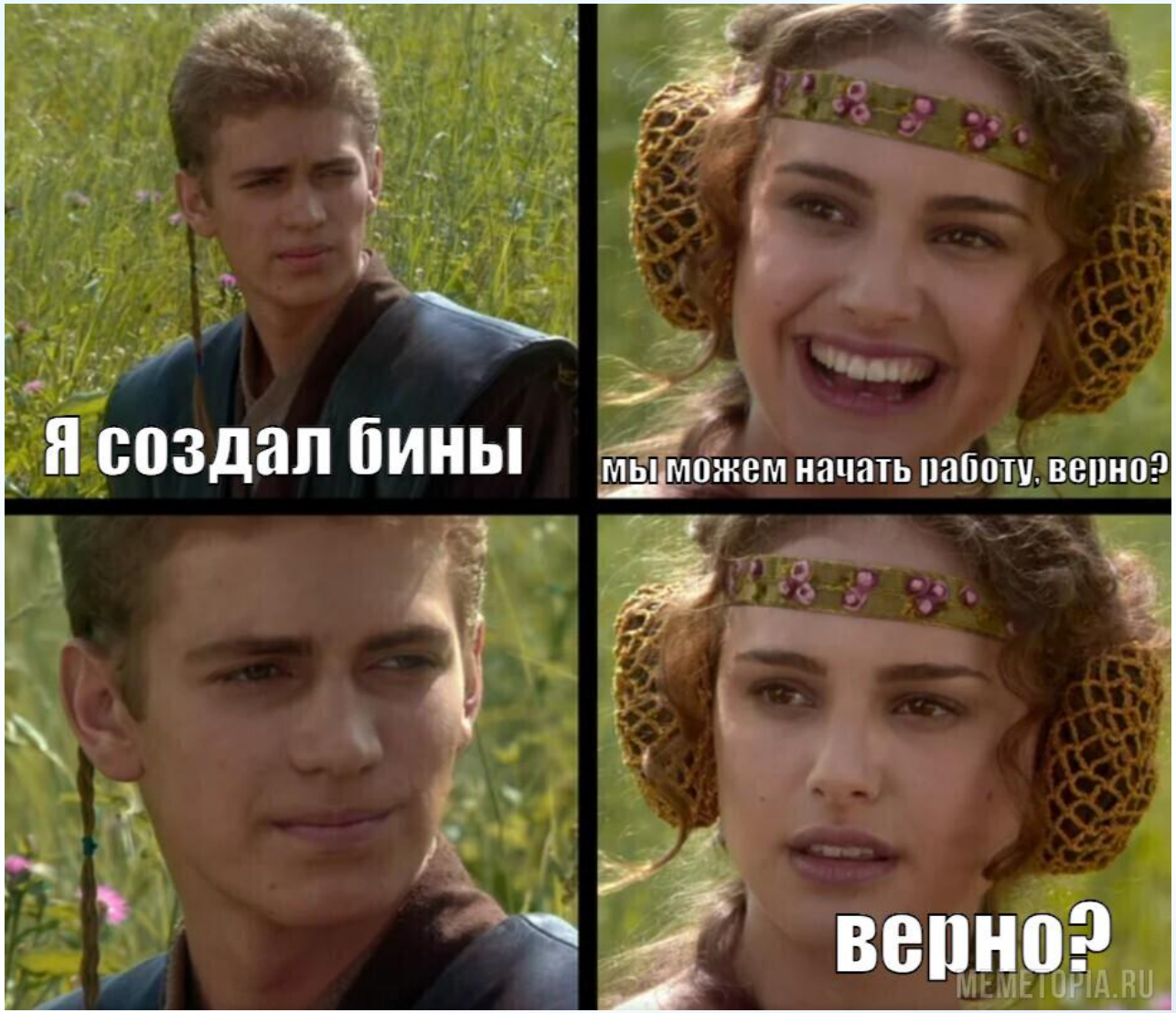
Этап 4. Создание экземпляров бинов

Созданием экземпляров бинов занимается **BeanFactory**, при необходимости делегируя самописным **FactoryBean**.

Экземпляры бинов создаются на основе **BeanDefinition**.

После создания бина, (*неважно бобовой фабрикой или самописной фабрикой бобов*), они почти попадают в **Map<BeanName, Bean>** великого и ужасного **ApplicationContext**.

Итоги четвертого этапа



spring

Этап 5. Настройка созданных бинов

Разработчик может модернизировать бины до того, как они попадут в мапу **ApplicationContext**, то есть чуть-чуть донастроить их.

BeanPostProcessor

Аналогично **BeanFactoryPostProcessor**, который дает разработчику доступ к созданным **BeanDefinition**, **BeanPostProcessor** дает разработчику доступ к созданным бинам.

```
public interface BeanPostProcessor {

        Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException;

        Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException;

}
```

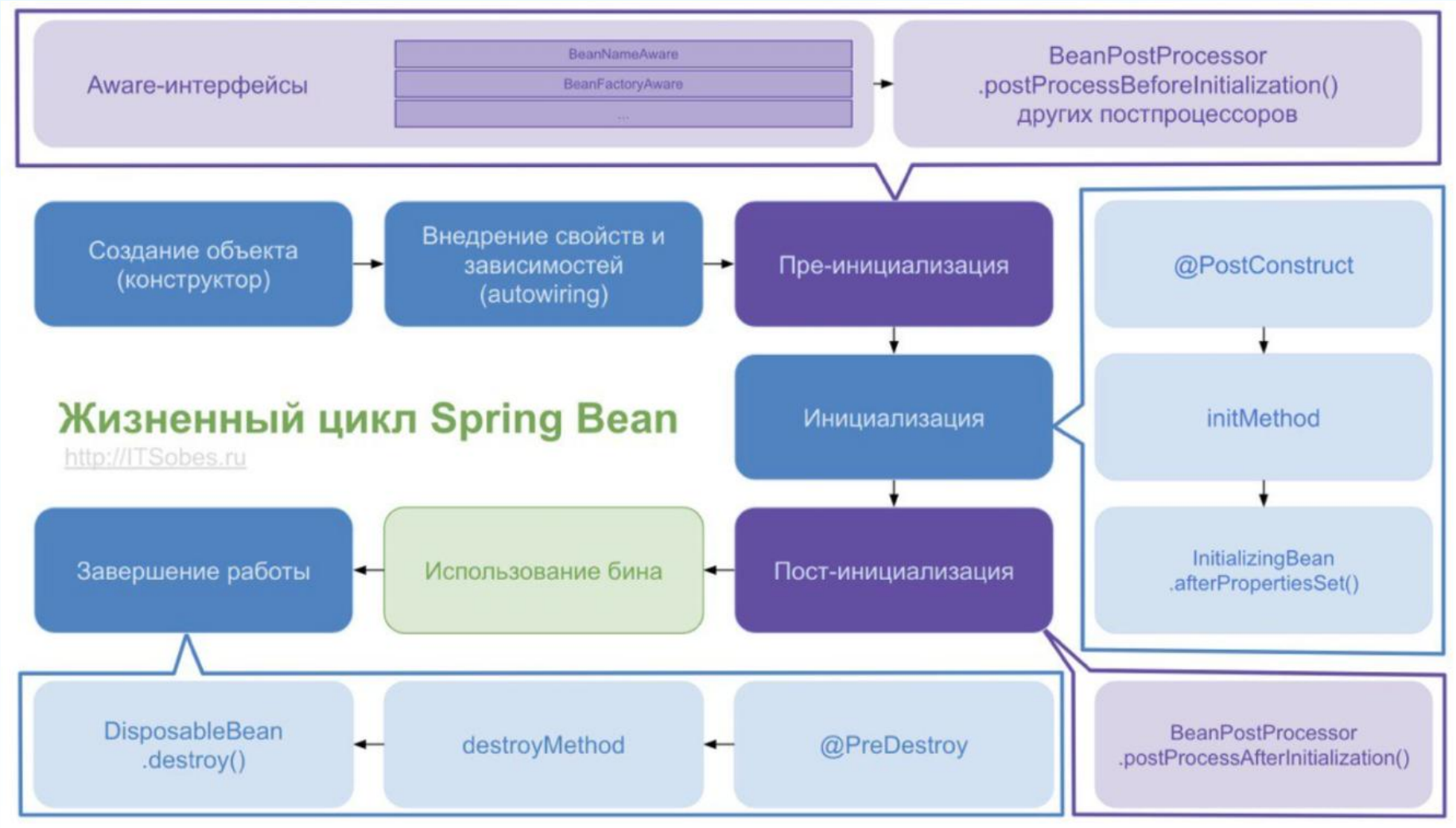
spring

BeanPostProcessor

Разница между методами в порядке их вызова, первый вызывается до инициализации бина, второй после.

- Оба метода обязаны вернуть в ответ бин, потому что через **BeanPostProcessor** проходят абсолютно все бины, даже, если с ними ничего не произойдет внутри, вы все равно получите на выходе все бины, в которых по ссылкам лежит null – значения.
- **BeanPostProcessor** прекарсно подходит для реализации прокси, (имейте ввиду, что прокси принято делать в методе **postProcessAfterInitialization**, то есть после инициализации бина).

Схема создания бинов



Пример

Есть аннотация, которой будут помечаться поля класса, в которые нужно записать значение.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface InjectRandomInt {
        int min() default 0;
        int max() default 10;
}
```

Пример

Необходима реализация BeanPostProcessor, которая будет обрабатывать эту аннотацию.

```
@Component
public class InjectRandomIntBeanPostProcessor implements BeanPostProcessor {
  @Override
  public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
    Field[] fields = bean.getClass().getDeclaredFields();
    for (Field field : fields) {
      if (field.isAnnotationPresent(InjectRandomInt.class)) {
      field.setAccessible(true);
      InjectRandomInt annotation = field.getAnnotation(InjectRandomInt.class);
      ReflectionUtils.setField(field, bean, getRandomIntInRange(annotation.min(), annotation.max()));
      }
    }
    return bean;
  }
```

spring

Пример

```
  @Override
  public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
    return bean;
  }

  private int getRandomIntInRange(int min, int max) {
    return min + (int)(Math.random() * ((max - min) + 1));
  }
}
```

BeanPostProcessor обязательно должен быть бином, поэтому мы его либо помечаем аннотацией @Component, либо регистрируем его в xml конфигурации как обычный бин.

Пример

В итоге, все бины типа MyBean будут создаваться с уже проинициализированными полями value1 и value2.

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class MyBean {

        @InjectRandomInt
        private int value1;

        @InjectRandomInt(min = 100, max = 200)
        private int value2;

        private int value3;
}
```

Итоги пятого этапа

Наконец-то Spring закончил работу.

Бины созданы со всеми нужными зависимостями и доработками разработчика.

ApplicationContext заполнен всеми бинами, которые разработчик хотел видеть в своем приложении.

Или в консоле появился stack trace с исключением…)

Spring Bean Scopes

Singleton – область видимости по умолчанию, один бин на весь runtime.

Prototype – возвращает новый экземпляр каждый раз, когда запрашивается из **ApplicationContext**.

Request – на каждый HTTP – запрос создается новый экземпляр.

Session – на каждую HTTP – сессию создается новый экземпляр.

Application – один на **ServletContext**.

Websocket – один на каждый **WebSocket**.

Что еще?

@Condition – https://www.baeldung.com/spring-conditional-annotations

@Qualifier – https://www.baeldung.com/spring-qualifier-annotation

@Primary – https://www.baeldung.com/spring-primary

@Value – https://www.baeldung.com/spring-value-annotation

Spring YAML Configuration – https://www.baeldung.com/spring-yaml

Многому научиться предстоит еще тебе, падаван. Но в руках меч уже держать спринговый можешь ты.