нетология

Dependency Lookup, Dependency Injection, IoC, Spring, Application Context

Григорий
Вахмистров

Григорий Вахмистров

Backend Developer в Tennisi.bet

План занятия

1. Предисловие
2. Dependencies
3. Dependency Injection
4. BeanFactory
5. Application Context
6. Annotation Config
7. Java Config
8. Итоги
9. Домашнее задание

Предисловие

Предисловие

На прошлой лекции мы посмотрели на разработку прототипа веб-приложения и пришли к выводу, что нам не хватает двух инструментов:

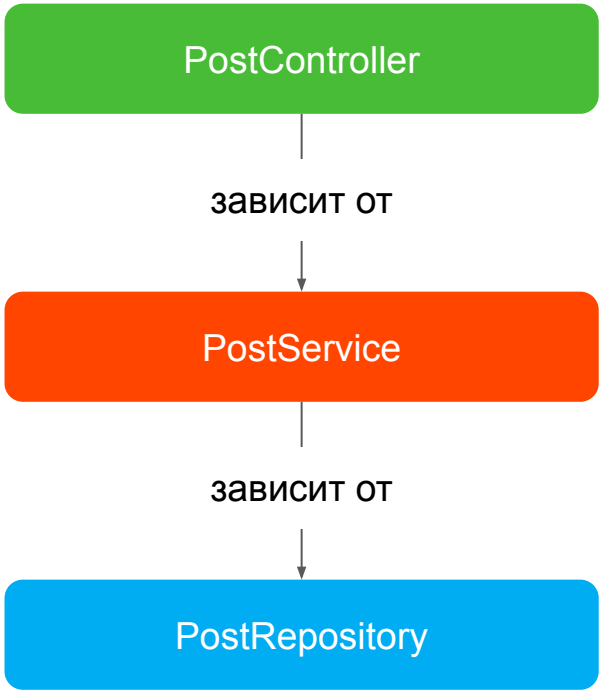
1. Инструмента "управления зависимостями"
2. Инструмента, упрощающего типовые операции с веб-запросами

Сегодня мы как раз займёмся рассмотрением первого пункта данного перечня.

Dependencies

Dependencies

Когда для работы одному объекту требуется другой - это называется зависимостью. Для проекта с прошлой нашей лекции:



Dependency Graph

Не сложно увидеть, что зависимости из себя представляют направленный граф (**граф зависимостей** - **dependency graph**).

Подходы к управлению зависимостями

Когда объекту нужна зависимость, есть несколько вариантов, её (зависимости) получения.

Вопрос к аудитории: *какие это варианты?*

Создание зависимости

Первый и самый простой - любой объект внутри себя может создать необходимую ему зависимость.

Плюсы:

- удобно
- просто

Минусы:

- сложно переиспользовать (между разными объектами)
- сложно тестировать (нельзя подставить заглушку)

Поиск зависимости (lookup)

Второй - создать объект реестр и передавать всем объектам ссылку на него (либо сделать синглтоном), в котором объекты могут искать зависимости.

Плюсы:

- просто реализовать

Минусы:

- объекты слишком много знают о своём окружении (завязаны на то, что должен существовать реестр)

Внедрение зависимости (injection)

Третий - написать такой сервис, который сам будет анализировать связи между объектами, создавать их и инжектировать необходимые зависимости.

Плюсы:

- объекты ничего не знают о существовании окружения, которое их создаёт и связывает (DI контейнер)

Минусы:

- сложно реализовать

Dependency Injection

Spring

Spring Framework (или просто Spring) - инструмент, предоставляющий DI контейнер и функциональность работы с контекстом приложения.

На сегодняшний день вокруг Spring существует целая экосистема связанных проектов, некоторые из которых вообще никакого отношения к DI и контейнерам не имеют.

Ключевая идея

Ключевая идея - использование компонентов, основанных на POJO (Plain Old Java Objects): объектах, которые не обязаны от кого бы то ни было наследоваться или реализовать какие-либо интерфейсы.

Пример: чтобы создать компонент, который работает в Servlet Container, мы обязаны отнаследоваться от HttpServlet (или реализовать интерфейс Servlet). То же самое относится и к другим компонентам. Spring же предлагает использовать в качестве компонентов любые объекты, не накладывая ограничений на разработчика.

Java EE vs Spring

Java EE:

- Контейнеры
- Компоненты
- Сервисы

Spring:

- Lightweight DI контейнер
- Компоненты

Стоит понимать, что Spring лишь ментально противопоставляет себя модели Java EE, при этом прекрасно используя стек Java EE, встраиваясь в него (увидим на следующей лекции).

IoC (Inversion of Control)

IoC - паттерн, обозначающий инверсию управления: объекты встраиваются в модель жизненного цикла фреймворка и реагируют на сообщения, посылаемые им (init, destroy, service и т.д.).

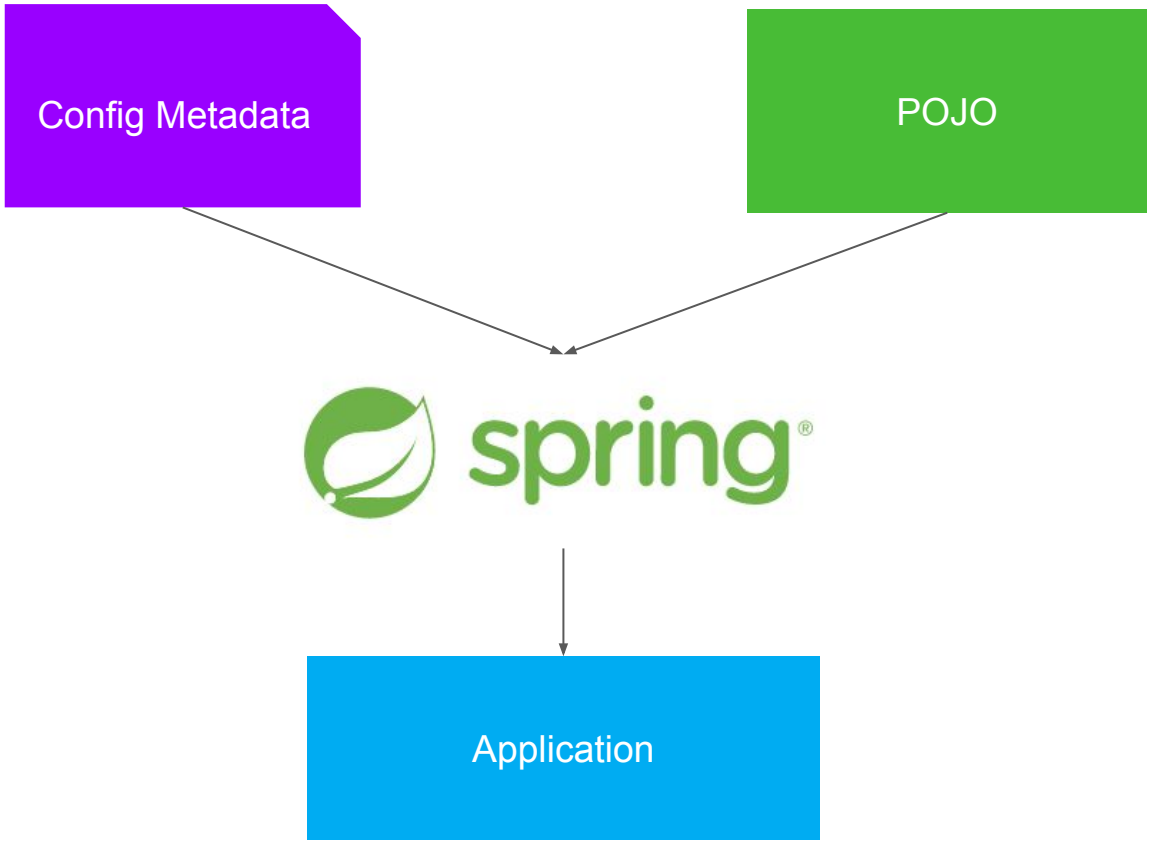
Несмотря на то, что сугубо технически DI и IoC это немного о разном (DI о разрешении зависимостей, IoC об инверсии управления), в документации Spring эти термины считаются взаимозаменяемыми, поэтому и мы в рамках курса будем их считать таковыми.

BeanFactory

Spring

```
<dependencies>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.1</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.9.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.8.6</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

Spring Container



О подходах к изучению

Стоит отметить, что на сегодняшний день Spring предоставляет настолько широкие возможности по собственной настройке и использованию реализованных механизмов, что только о них (не приступая к какому-либо практическому использованию) можно рассказывать несколько десятков лекций.

На начальных этапах изучения это бессмысленно, поскольку наша задача - научиться использовать Spring именно в практических целях, пользуясь наиболее распространёнными подходами к его настройке.

О подходах к изучению

Поэтому всё, что будет говориться в лекциях по Spring - это лишь малая часть возможностей. И там, где говорится, что "можно делать вот так", можно делать ещё как минимум парой других способов, а к любым правилам почти всегда предоставляются исключения.

BeanFactory & Beans

Ключевой интерфейс, который и является контейнером - BeanFactory (фабрика бинов).

Spring Bean (не путать с Java Bean) - это объект, управляемый контейнером (т.е. контейнер решает, когда создавать этот объект, когда уничтожать, как инжектировать зависимости).

Beans

В качестве бинов обычно определяют "сервисные" объекты, которые должны существовать в вашем приложении в единственном (редко - нескольких) экземплярах.

Не нужно делать бинами модели, exception'ы и подобные объекты (хотя вы вполне можете столкнуться с бинами, которые существуют только в рамках обработки конкретного HTTP-запроса).

Bean Configuration

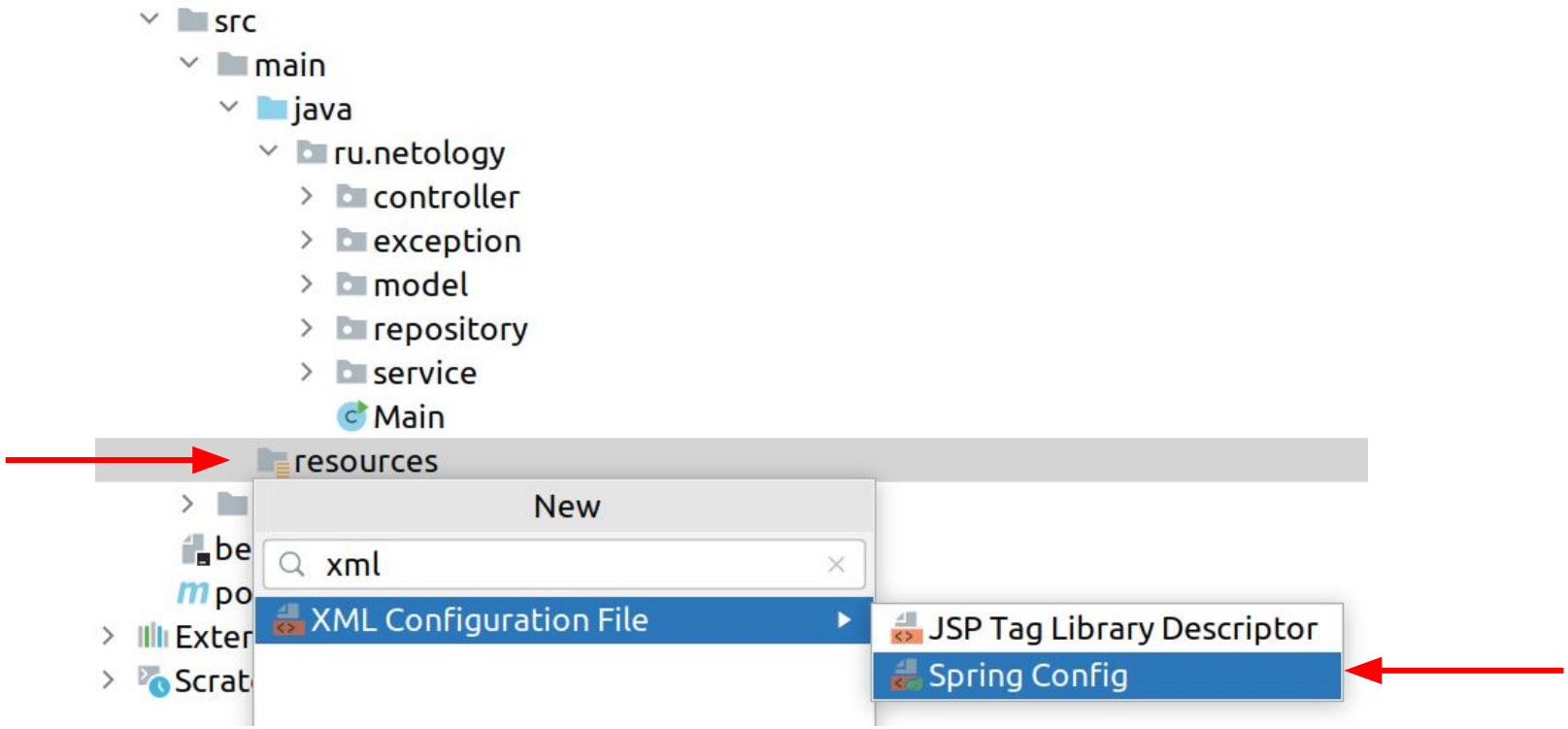
Spring предоставляет достаточно много способов конфигурации бинов:

- XML
- Annotation
- Java
- Groovy
- Kotlin
- Programmatic (программное регистрирование бинов)

Самыми распространёнными являются Annotation и Java. XML - это то, с чего всё начиналось, поэтому мы начнём с него и продолжим уже с Annotation и Java Config.

XML Config

Механика: создаётся конфигурационный файл, в котором описываются определения бинов. Этот файл обрабатывается Reader'ом, умеющим XML превращать в Java-объекты (Bean Definition'ы). Bean Definition'ы загружаются в контейнер (принцип разделения обязанностей):



XML Config

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/
    <bean id="postController" class="ru.netology.controller.PostController" autowire="constructor" />
    <bean id="postService" class="ru.netology.service.PostService" autowire="constructor" />
    <bean id="postRepository" class="ru.netology.repository.PostRepository" />
</beans>
```

Каждая строка - это определения бина (Bean Definition). Класс указывается для того, чтобы Spring средствами Reflection API мог создавать объекты (бины). autowire="constructor" означает, что Spring будет автоматически связывать бины друг с другом через параметры конструктора.

То, что прописано в id, является именем бина (дополнительные имена можно задать через атрибут name).

XML Config

```
7  public class Main {
8    public static void main(String[] args) {  args: {}
9      final var factory = new DefaultListableBeanFactory();  factory: DefaultListableBeanFactory@1631
10     final var reader = new XmlBeanDefinitionReader(factory);  reader: XmlBeanDefinitionReader@1632
11     reader.loadBeanDefinitions( location: "beans.xml");  reader: XmlBeanDefinitionReader@1632
12
13     // получаем по имени бина
14     final var controller : Object  = factory.getBean( name: "postController");  controller: PostController@1693
15
16     // получаем по классу бина
17     final var service : PostService  = factory.getBean(PostService.class);  factory: DefaultListableBeanFactory@1631
18
19     // по умолчанию создаётся лишь один объект на BeanDefinition
20     final var isSame = service == factory.getBean( name: "postService");
21   }
22 }
```

В дебаггере:

scope

Нужно отдельно остановиться на правиле один bean definition - один объект. Свойство, определяющее подобное поведение называется scope. По умолчанию оно имеет значение singleton (и используется в большинстве случаев).

Помимо singleton могут быть ещё и prototype (в webmvc и другие), который при каждом вызове создаёт и возвращает новый объект.

autowiring

Благодаря autowire, нам потребовалось лишь "закинуть" определения бинов в контейнер, всё остальное он взял на себя.

XML Config

В классическом xml config обычно напрямую связывали бины без autowire (это позволяло разделять этапы сборки приложения и конфигурирования - можно было без перекомпиляции переконфигурировать приложение):

```
<bean id="postController" class="ru.netology.controller.PostController">
    <constructor-arg name="service" ref="postService" />
</bean>
<bean id="postService" class="ru.netology.service.PostService">
    <constructor-arg name="repository" ref="postRepository" />
</bean>
<bean id="postRepository" class="ru.netology.repository.PostRepository" />
```

autowiring

Изначально autowiring существовал только в следующих формах:

- по аргументу конструктора;
- по property (напоминаем, что property - это private field + get/set).

С активным использованием механизмов Reflection API стало возможным (не в XML) инжектирование прямо в поля (без property), но в большинстве случаев это считается плохим тоном для кода приложения, но при этом активно используется в автотестах.

autowiring

В большинстве случаев хорошей практикой считается DI через конструктор в финальное поле, поскольку этим вы явно показываете, что объект данного класса не может существовать без зависимости.

Если конструктор у такого класса всего один, то он и будет использован Spring'ом.

Ключевой бонус

Spring старается поддерживать максимально правильные способы разработки (снаружи), поэтому предоставляет возможность производить инжектирование и по типу интерфейса (а не конкретной реализации).

Давайте смотреть это сразу на реализации с аннотациями, а не XML Config.

Application Context

Application Context

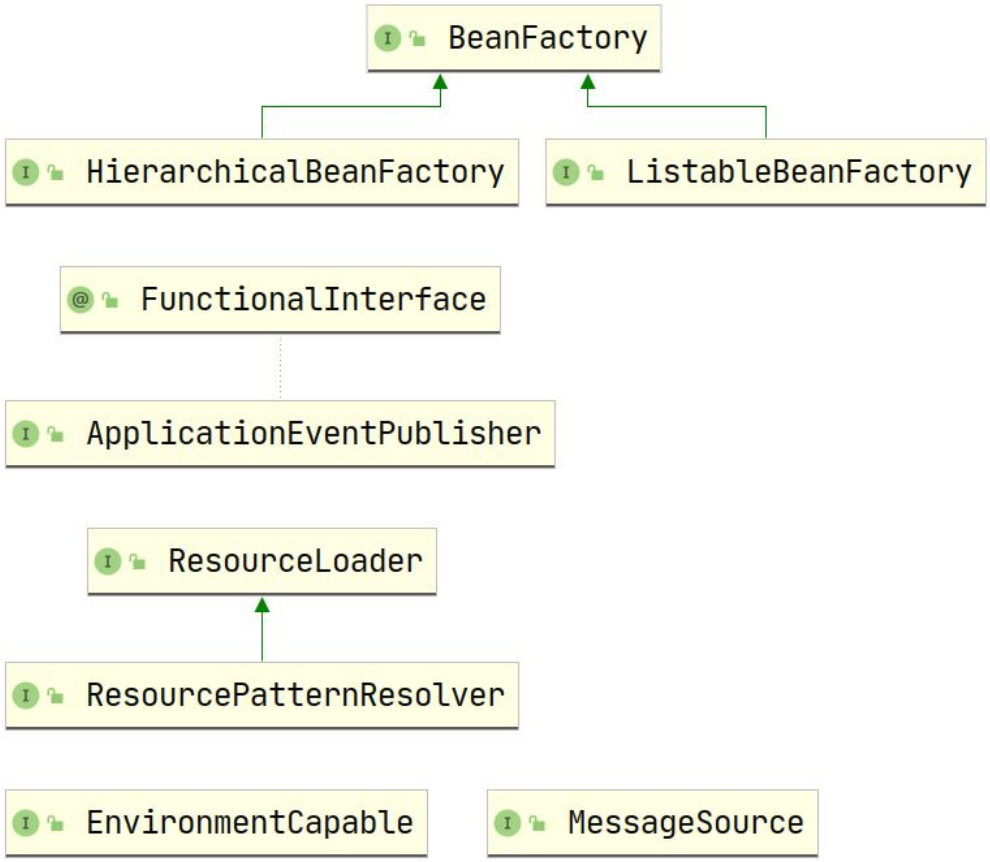
BeanFactory (именно DefaultListableBeanFactory) напрямую используется достаточно редко (только если вы хотите от Spring'а самый минимум - DI).

В основном используется ApplicationContext - это интерфейс, описывающий объекты, которые предоставляют приложению доп. возможности помимо DI:

- управление ресурсами
- локализацию
- внутреннюю шину сообщений
- и т.д.

Application Context

Интерфейсы, наследуемые ApplicationContext:



Application Context

ApplicationContext является наследником BeanFactory (на самом деле, в большинстве реализаций просто содержит внутри BeanFactory, которой и делегирует все запросы):

```
public Object getBean(String name) throws BeansException {
  this.assertBeanFactoryActive();
  return this.getBeanFactory().getBean(name);
}
```

Annotation Config

Annotation Config

Annotation Config подразумевает, что вы используете аннотации @Component (и производные), помечая классы из которых необходимо создавать бины.

Помимо @Component есть @Service и @Repository, @Controller, поэтому мы сразу будем использовать специализированные.

Annotation Config

```java
@Controller
public class PostController {
  public static final String APPLICATION_JSON = "application/json";
  private final PostService service;

  public PostController(PostService service) { this.service = service; }
```

```java
@Service
public class PostService {
  // сервис завязан на интерфейс, а не на конкретную реализацию
  private final PostRepository repository;

  public PostService(PostRepository repository) { this.repository = repository; }
```

```java
@Repository
public class PostRepositoryStubImpl implements PostRepository {
```

Annotation Config

```
public static void main(String[] args) {
  // отдаём список пакетов, в которых нужно искать аннотированные классы
  final var context = new AnnotationConfigApplicationContext( ...basePackages: "ru.netology");

  // получаем по имени бина
  final var controller : Object  = context.getBean( name: "postController");

  // получаем по классу бина
  final var service : PostService  = context.getBean(PostService.class);

  // по умолчанию создаётся лишь один объект на BeanDefinition
  final var isSame = service == context.getBean( name: "postService");
}
```

В случае использования аннотаций именем бина становится имя класса с переведённой в нижний регистр первой буквой.

@Autowired

В Spring существует аннотация @Autowired, которой можно помечать конструкторы, setter'ы и поля, показывая Spring, куда нужно осуществить подстановку зависимости.

Если вы осуществляете DI через конструктор и конструктор всего один, то написание данной аннотации не требуется.

Типичные ошибки

При работе со Spring пользователи допускают ряд типичных ошибок:

1. Не предоставляют все необходимые зависимости
2. Предоставляют несколько зависимостей одного типа
3. Создают циклические зависимости

Во всех этих сценариях Spring формирует достаточно информативные Exception'ы, которые обязательно нужно читать целиком.

ApplicationContext vs BeanFactory

Между **ApplicationContext** и **BeanFactory*** существует ключевая разница: BeanFactory ленива - пока вы у неё не попросите Bean, она его не создаст, ApplicationContext после конфигурирования запускает (либо вы запускаете) метод refresh, в котором настраивает ключевые свои сервисы и заставляет BeanFactory проинициализировать бины.

Примечание:* речь идёт конкретно о наследниках AbstractApplicationContext и AbstractBeanFactory.

refresh

```
@Override
public void refresh() throws BeansException, IllegalStateException {
  synchronized (this.startupShutdownMonitor) {
    // Prepare this context for refreshing.
    prepareRefresh();

    // Tell the subclass to refresh the internal bean factory.
    ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

    // Prepare the bean factory for use in this context.
    prepareBeanFactory(beanFactory);

    try {...}

    catch (BeansException ex) {...}

    finally {...}
  }
}
```

refresh

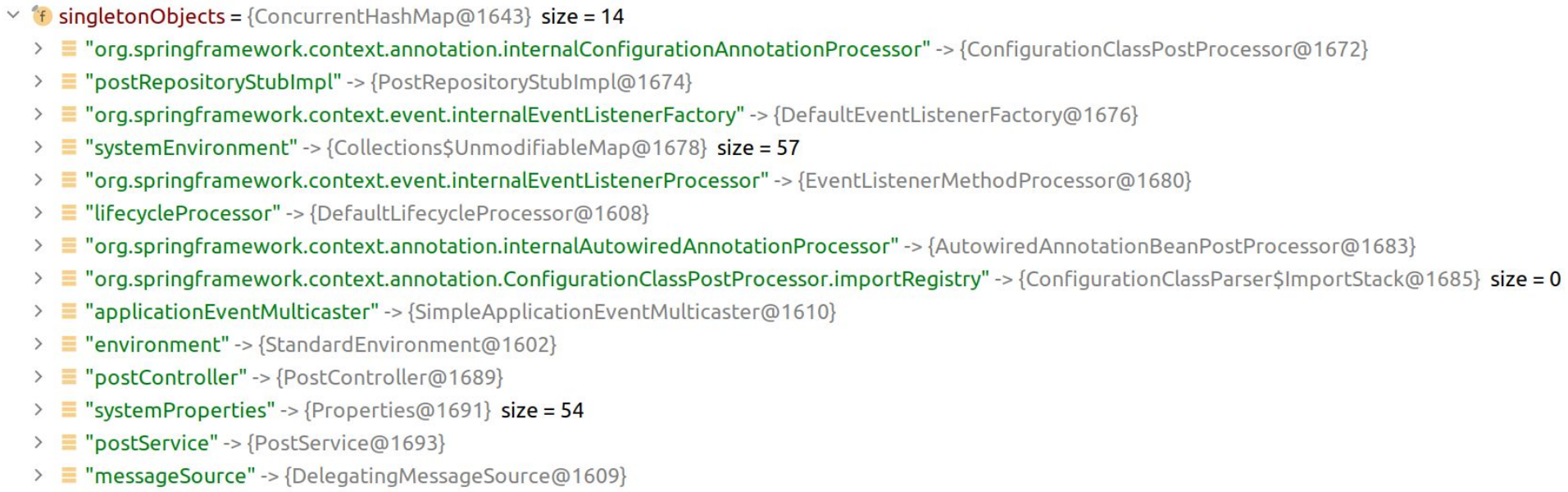
```
// Allows post-processing of the bean factory in context subclasses.
postProcessBeanFactory(beanFactory);

// Invoke factory processors registered as beans in the context.
invokeBeanFactoryPostProcessors(beanFactory);

// Register bean processors that intercept bean creation.
registerBeanPostProcessors(beanFactory);

// Initialize message source for this context.
initMessageSource();

// Initialize event multicaster for this context.
initApplicationEventMulticaster();

// Initialize other special beans in specific context subclasses.
onRefresh();

// Check for listener beans and register them.
registerListeners();

// Instantiate all remaining (non-lazy-init) singletons.
finishBeanFactoryInitialization(beanFactory);

// Last step: publish corresponding event.
finishRefresh();
```

ApplicationContext vs BeanFactory

Кроме того, ApplicationContext позволяет подписываться на "завершение работы" (где-то мы это уже видели):

```
@Override
public void registerShutdownHook() {
  if (this.shutdownHook == null) {
    // No shutdown hook registered yet.
    this.shutdownHook = new Thread(SHUTDOWN_HOOK_THREAD_NAME) {
      @Override
      public void run() {
        synchronized (startupShutdownMonitor) {
          doClose();
        }
      }
    };
    Runtime.getRuntime().addShutdownHook(this.shutdownHook);
  }
}
```

BeanFactory внутри



Приведён скриншот поля BeanFactory, содержащейся внутри AnnotationConfigApplicationContext'а.

Annotation Config

Annotation Config хорош для ситуации, аналогичной нашей - нам из каждого класса нужно по одному бину.

Но не всегда данный подход применим: мы не можем написать аннотации над тем кодом, который не контролируем (например, создать из классов, входящих в Spring бины).

В этом случае нам поможет Java Config.

Java Config

Java Config

Java Config предполагает создание специального конфигурационного класса, помеченного @Configuration. Этот класс специальным образом "разбирается" в рантайме таким образом, что из его методов, помеченных аннотацией @Bean, создаются Bean'ы и Bean Definition'ы, а все вызовы к этим методом перехватываются (паттерн Proxy) и заменяются на dependency injection.

При этом аннотации над самими классами не ставятся* (в отличие от Annotation Config).

Примечание:* речь идёт чистом Annotation Config.

Java Config (первый вариант)

```java
@Configuration
public class JavaConfig {
  @Bean
  // аргумент метода и есть DI
  // название метода - название бина
  public PostController postController(PostService service) {
    return new PostController(service);
  }

  @Bean
  public PostService postService(PostRepository repository) {
    return new PostService(repository);
  }

  @Bean
  public PostRepository postRepository() {
    return new PostRepositoryStubImpl();
  }
}
```

Context

```
public class Main {
    public static void main(String[] args) {
        // отдаём класс конфигурации
        final var context = new AnnotationConfigApplicationContext(JavaConfig.class);

        // получаем по имени бина
        final var controller : Object  = context.getBean( name: "postController");

        // получаем по классу бина
        final var service : PostService  = context.getBean(PostService.class);

        // по умолчанию создаётся лишь один объект на BeanDefinition
        final var isSame = service == context.getBean( name: "postService");
    }
}
```

Java Config (второй вариант)

```java
@Configuration
public class JavaConfig {
  @Bean
  // название метода - название бина
  public PostController postController() {
    // вызов метода и есть DI
    return new PostController(postService());
  }

  @Bean
  public PostService postService() {
    return new PostService(postRepository());
  }

  @Bean
  public PostRepository postRepository() {
    return new PostRepositoryStubImpl();
  }
}
```

Java Config

Чем хорош Java Config:

1. вы можете использовать логику при инициализации и дебажить (XML и аннотации дебажить достаточно проблематично);
2. вся конфигурация хранится в одном файле.

В реальной жизни

В реальной жизни чаще всего используется микс из вариантов конфигурации:

- те бины, для инициализации которых не нужна логика и над которыми у вас есть контроль, помечаются @Component
- там, где логика нужна или у вас нет контроля над классами - Java Config
- там, где логика не нужна или у вас нет контроля над классами или вы хотите иметь возможность переконфигурировать приложение без перекомпиляции - XML Config

Итоги

Итоги

Сегодня мы посмотрели на основы Spring, узнали, что такое DI и посмотрели на основы его использования.

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** мессенджера .
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Григорий Вахмистров