Оглавление

[Урок 1: Введение 5](#_Toc28124269)

[Что такое framework? 5](#_Toc28124270)

[Spring Framework 5](#_Toc28124271)

[А можно ли без Spring? 7](#_Toc28124272)

[Application Context (AC) и Dependency Injection (DI) 8](#_Toc28124273)

[Удобный и эффективный доступ к БД 9](#_Toc28124274)

[Spring MVC 9](#_Toc28124275)

[Другие компоненты Spring Framework 10](#_Toc28124276)

[Урок 4: Inversion of Control (IoC) 11](#_Toc28124277)

[Bean 12](#_Toc28124278)

[IoC 13](#_Toc28124279)

[Урок 5: DI. Введение 15](#_Toc28124280)

[Типичные шаги в работе со Spring 15](#_Toc28124281)

[Способы внедрения зависимостей 15](#_Toc28124282)

[Урок 6: DI. Часть 2 17](#_Toc28124283)

[Способы внедрения зависимостей 17](#_Toc28124284)

[Внедрение зависимостей через setter 18](#_Toc28124285)

[Внедрение простых значений 19](#_Toc28124286)

[Внедрение значений из внешнего файла 19](#_Toc28124287)

[Урок 7: Bean Scope («Область видимости» бинов) 20](#_Toc28124288)

[Способы внедрения зависимостей 20](#_Toc28124289)

[Что такое scope? 20](#_Toc28124290)

[Singleton 20](#_Toc28124291)

[Prototype 22](#_Toc28124292)

[Другие scope 22](#_Toc28124293)

[Урок 8: Жизненный цикл бина. Init, Destroy и Factory методы. 23](#_Toc28124294)

[Способы внедрения зависимостей 23](#_Toc28124295)

[Жизненный цикл бина (Bean Lifecycle) 23](#_Toc28124296)

[init и destoy методы 24](#_Toc28124297)

[Тонкости init и destroy методов 24](#_Toc28124298)

[factory‑method 25](#_Toc28124299)

[Урок 9: Аннотации. Введение 27](#_Toc28124300)

[Что такое Java Аннотации? 27](#_Toc28124301)

[Зачем использовать аннотации? 27](#_Toc28124302)

[Как работает конфигурация с помощью аннотаций? 27](#_Toc28124303)

[Аннотация @Component 28](#_Toc28124304)

[Урок 10: Аннотация @Autowired. DI 29](#_Toc28124305)

[Аннотация @Autowired 29](#_Toc28124306)

[Как работает аннотация @Autowired? 29](#_Toc28124307)

[Еще об @Autowired 30](#_Toc28124308)

[Урок 11: Аннотация @Qualifier. DI 33](#_Toc28124309)

[Аннотация @Qualifier 33](#_Toc28124310)

[Необычный синтаксис 33](#_Toc28124311)

[Урок 12: Аннотации @Scope, @Value, @PostConstruct, @PreDestroy 35](#_Toc28124312)

[Внедрение значения из внешнего файла 35](#_Toc28124313)

[Scope 36](#_Toc28124314)

[Жизненный цикл бина (Bean Lifecycle) 37](#_Toc28124315)

[init и destoy методы 37](#_Toc28124316)

[Аннотации @PostConstruct и @PreDestroy 38](#_Toc28124317)

[Сигнатура @PostConstruct и @PreDestroy методов 39](#_Toc28124318)

[Урок 13: Конфигурация с помощью Java кода 40](#_Toc28124319)

[Аннотация @Configuration 41](#_Toc28124320)

[Для каждого XML тега есть соответствующая аннотация 42](#_Toc28124321)

[Использование конфигурационного файла 43](#_Toc28124322)

[Ручное внедрение зависимостей (без @Autowired) с помощью Java конфигурации 43](#_Toc28124323)

[Побробнее об аннотации @Bean 44](#_Toc28124324)

[Внедрение значений из внешнего файла 45](#_Toc28124325)

[Урок 14: Spring MVC. Теория 46](#_Toc28124326)

[Что такое Spring MVC 47](#_Toc28124327)

[MVC (Model‑View‑Controller) 47](#_Toc28124328)

[Из чего состоит Spring MVC приложение 48](#_Toc28124329)

[DispatcherServet 48](#_Toc28124330)

[Контроллер (Controller) 49](#_Toc28124331)

[Модель (Model) 49](#_Toc28124332)

[Представление (View) 49](#_Toc28124333)

[Урок 15. Spring MVC. Первое приложение 50](#_Toc28124334)

[Что мы сделаем на этом уроке? 50](#_Toc28124335)

Урок 1: Введение

Что такое framework?

С англ. — каркас, структура

Платформа которая определяет структуру приложения и облегчает разработку компонентов системы и их интеграцию.

Фреймворк — это больше, чем просто библиотека (определяет структуру системы, предоставляет определенный паттерны разработки)

Популярные фреймворки

Spring Framework, Node.js, Django, Ruby on Rails, ASP.NET и множество других.

Spring Framework

История

|  |  |
| --- | --- |
| Версия | Год |
| 0.9  1.0  2.0  2.5  3.0  3.1  4.0  4.2.0  4.3  5 | 2003  2004  2006  2007  2009  2011  2013  2015  2016  2017 |

Востребованность Spring

Один из самых популярных web‑фреймворков в мире

Самый популярный Java‑фреймворк

Java — один из самых популярных ЯП в мире. Spring обычно используется везде, где используется Java

Очень востребован работодателями по всему миру — достаточно зайти на любой сайт по поиску работы

Spring Framework

Состоит из множества компонентов, которые облегчает множество аспектов разработки приложений на Java

1. Контекст приложения (Application Context) и Внедрение Зависимостей (Dependency Injection)
2. Удобный и эффективный доступ к БД (замена JDBC)
3. Компонент для разработки Web — приложения на Java (Spring MVC)
4. Множество других полезных компонентов (spring.io)

Application Context и Dependency Injection

Типичное Java приложение — это набор Java объектов, которые взаимодействуют друг с другом и ссылаются друг на друга. Чем больше и сложнее приложение, тем больше объектов и сложнее иерархия между ними. Когда Java приложение запускается, все необходимые Java объекты создаются и помещаются в оперативную память. В ходе работы приложения, объекты могут добавляться или удаляться. Также могут изменяться связи между объектами. Большое количество объектов и связей между ними встречается в любом более‑менее сложном Java‑приложении. Spring помогает нам в работе с множеством объектов.

А можно ли без Spring?

Можно, но сложно — создание объектов вручную

1. Код становится очень запутанным (особенно в больших приложениях), не прослеживается иерархия между объектами
2. Engine engine = new Engine();

У класса Engine тоже могут быть объекты, которые ему нужны для работы, и так далее

1. Много лишнего кода

Другой пример:

Часто нам необходимо один объект делить между всеми другими объектами. Этот объект должен создаваться только один раз.

Нам надо:

1. сделать так, чтобы объект этого класса создавался максимум один раз (нам не нужны лишние соединения с базой данных)
2. внедрить ссылку на объект этого класса всем остальным классам, которым надо работать с БД

Проблема 1

Сделать так, чтобы объекты данного класса создавался максимум один раз.

Можно решить с помощью паттерна Singlton, но для этого требуется дополнительный код.

Решить проблему с помощью Spring.

Проблема 2

Внедрить зависимость на данный объект во все остальные классы, выстроить иерархию.

Можно вручную — сложно, запутано.

Решить проблему с помощью Spring.

Application Context (AC) и Dependency Injection (DI)

AC

Мы описываем объекты, которые необходимы для работы нашего приложения (которые должны создаваться при запуске приложения)

Spring сам создает эти объекты и берет на себя управление этими объектами (их жизненный цикл и многое другое)

DI

Spring сам внедряет все необходимые зависимости в объекты (связывает объекты между собой). Нам необходимо только описать эту связь, Spring делает все остальное. Это называется DI

AC и DI

Важная часть Spring Framework

Будет подробно разбираться в ходе курса.

Удобный и эффективный доступ к БД

Большинство приложений используют БД.

JDBC — примитивный и неудобный способ взаимодействия с БД.

JDBC не походит для сложных приложений. Он слишком низкоуровневый.

Spring предоставляет множество инструментов для взаимодействия с БД.

Spring MVC

Компонент Spring Framework, который позволяет создавать Web‑приложения.

Огромное количество Web‑приложений в интернете работают на Spring MVC.

Помимо этого, Spring MVC часто используется в качестве backend‑API для мобильных приложений.

Spring MVC > Java EE Servlets

Другие компоненты Spring Framework

Spring Security, Spring Boot, Spring Webflow и множество других

spring.io/projects

Урок 4: Inversion of Control (IoC)

Проблема

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | class ClassicalMusic {  // код для доступа к классической музыке  }  class MusicPlayer {  private ClassicalMusic classicalMusic;  public void playMusic() {  classicalMusic  = new ClassicalMusic();  // код воспроизведения музыки  }  } |

Зависимости появляются очень часто в коде.

Проблема 1

MusicPlayer сильно зависит от ClassicalMusic, MusicPlayer «заточен» только на работу с ClassicalMusic.

Решение: использовать интерфейс (или абстрактный класс), который бы обобщал различные музыкальные жанры.

Решение

|  |  |
| --- | --- |
| 1.  2.  3.  4. | interface Music {  // код, который необходим для доступа  // к любому жанру музыки  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21. | class ClassicalMusic  implements Music {  // код для доступа к классической музыке  }  class RockMusic implements Music {  // код для доступа к рок музыке  }  class MusicPlayer {  // слабая зависимость  private Music music;  public void playMusic() {  music = new ClassicalMusic();  // или  music = new RockMusic();  // код для воспроизведения музыки  }  } |

Проблема 2

Объекты создаются вручную. Мы хотим вынести эти детали в конфигурационный файл, а не лезть каждый раз в код (и перекомпилировать его) для того, чтобы поменять объект.

Решение: Использовать Spring Framework, который сам создаст необходимые объекты (бины) согласно конфигурационному файлу.

Bean

Это просто Java объекты

Когда Java объекты создаются с помощью Spring, они называются бинами (beans)

Бины создаются из Java классов (так же, как и обычные объекты).

|  |  |
| --- | --- |
| 1.  2.  3.  4. | <bean id="testBean"  class="springFramework.TestBean">  <constructor-arg value="Neil"/>  </bean> |

Проблема 3

MusicPlayer сам создает свои зависимости. Это архитектурно неправильно — противоречит принципу IoC.

Решение: Использовать принцип IoC.

IoC

MusicPlayer зависит от ClassicalMusic

MusicPlayer сам создает объекты ClassicalMusic.

Вместо этого мы хотим передать объекты ClassicalMusic внутрь MusicPlayer — это и называется инверсией управления (IoC)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | class MusicPlayer {  private Music music;  // зависимость внедряется извне (IoC)  public MusicPlayer(Music music) {  this.music = music;  }  public void playMusic() {  // больше не создаем объекты!  // код для воспроизведения музыки  }  } |

Программируем на уровне интерфейсов — хороший архитектурный паттерн.

Последняя проблема

Объект, который мы хотим внедрить в MusicPlayer необходимо где‑то создать.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | public static void main(String[] args) {  MusicPlayer musicPlayer  = new MusicPlayer(  new RockMusic());  } |

Эту проблему мы решим на следующем уроке с помощью DI. Этой задачей тоже заниматься Spring Framework.

Перед тем как приступить к практике

Spring можно конфигурировать с помощью:

1. XML файла конфигурации (старый способ, но многие существующие приложения до сих пор его используют);
2. Java аннотации и немного XML (современный способ);
3. Все конфигурации на Java коде (современный способ).

Урок 5: DI. Введение

На прошлом уроке

Решили некоторые проблемы, связанные с зависимостями c помощью IoC

|  |  |
| --- | --- |
| 1.  2.  3.  4. | Music music = context.getBean("musicBean",  Music.class);  MusicPlayer musicPlayer  = new MusicPlayer(music); |

Но мы все еще внедряем зависимость music вручную

И вручную создаем объекты musicPlayer

Типичные шаги в работе со Spring

Создаем Java‑классы (будущие бины);

Создаем и связываем бины с помощью Spring (аннотации, XML или Java код);

При использовании, все объекты (бины) берутся из контейнера Spring.

Способы внедрения зависимостей

1. Через конструктор
2. Через setter
3. Есть множество конфигураций того, как внедрять (scope, factory method и т.д);
4. Можно внедрять через XML, аннотации или Java‑код;
5. Процесс внедрения можно автоматизировать (Autowiring)

На этом уроке внедрим через конструктор с использованием XML

На следующих — разберем все детали

Урок 6: DI. Часть 2

На прошлом уроке

Применили DI с помощью конструктора

Использовали XML‑конфигурацию

Внедрение (Injection) с помощью конструктора:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | <bean id="musicBean"  class="springFramework.ClassicalMusic">  </bean>  <bean id="musicPlayer"  class="springFramework.MusicPlayer">  <constructor-arg ref="musicBean"/>  </bean> |

Использование:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | MusicPlayer musicPlayer  = context.getBean("musicPlayer",  MusicPlayer.class);  musicPlayer.playMusic(); |

Способы внедрения зависимостей

Через конструктор

Через setter

Можем внедрять ссылки или простые значения

Можем внедрять значения из внешнего файла

Есть множество конфигураций (scope, init‑method, destroy‑method, factory method и т.д.)

Можно внедрять через XML, аннотации или Java‑код

Процесс внедрения можно автоматизировать (Autowiring)

Внедрение зависимостей через setter

Через конструктор:

|  |  |
| --- | --- |
| 1.  2.  3. | public MusicPlayer(Music music) {  this.music = music;  } |
| 1.  2.  3.  4. | <bean id="musicPlayer"  class="springFramework.MusicPlayer">  <constructor-arg ref="musicBean"/>  </bean> |

Через setter:

|  |  |
| --- | --- |
| 1.  2.  3. | public void setMusic(Music music) {  this.music = music;  } |
| 1.  2.  3.  4. | <bean id="musicPlayer"  class="springFramework.MusicPlayer">  <property name="music" ref="musicBean"/>  </bean> |

Setter

Setter — это «специальные методы», которые нужны для присвоения значений полям. Они позволяют нам не нарушать инкапсуляцию.

Внедрение простых значений

На всех прошлых уроках мы внедряли ссылки

Что если мы хотим внедрить не ссылку на другой объект, а какое‑то значение (число или строку)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | private String name;  private int vilume;  public void setName(String name) {  this.name = name;  }  public void SetVolume(int volume) {  this.volume = volume;  } |
| 1.  2. | <property name="name" value="Some name"/>  <property name="volume" value="50"/> |

Внедрение значений из внешнего файла

Не хотим каждый раз лезть в aplicationContext.xml;

Хотим все простые значения указать в одном файле.

|  |  |
| --- | --- |
| 1.  2. | musicPlayer.properties  musicPlayer.name=Some name  musicPlayer.volume=60 |
| 1.  2.  3.  4. | <property name="name"  value="${musicPlayer.name}"/>  <property name="volume"  value="${musicPlayer.volume}"/> |
| 1.  2. | <context:property-placeholder location  ="classpath:musicPlayer.properties"> |

Урок 7: Bean Scope («Область видимости» бинов)

Способы внедрения зависимостей

Через конструктор

Через setter

Можем внедрять ссылки или простые значения

Можем внедрять значения из внешнего файла

У бинов есть множество конфигураций (scope, init‑method, destroy‑method, factory method и т.д.)

Можно внедрять через XML, аннотации или Java‑код

Процесс внедрения можно автоматизировать (Autowiring)

Что такое scope?

scope задает то, как Spring будет создавать ваши бины.

|  |  |
| --- | --- |
| 1.  2.  3. | <bean id="musicBean"  class="springFramework.ClassicalMusic">  </bean> |

Лучше сразу разобрать на примере

Singleton

Scope, который используется по умолчанию.

|  |  |
| --- | --- |
| 1.  2.  3. | <bean id="musicBean"  class="springFramework.ClassicalMusic">  </bean> |

По умолчанию создается один объект (он создается до вызова метода getBean())

При всех вызовах getBean() возвращается ссылка на один и тот же единственный объект.

|  |  |
| --- | --- |
| 1.  2.  3. | MusicPlayer firstMusicPlayer  = context.getBean("musicPlayer",  MusicPlayer.class); |
| 1.  2.  3. | MusicPlayer secondMusicPlayer  = context.getBean("musicPlayer",  MusicPlayer.class); |

Создается единственный объект и возвращается ссылка на один и тот же объект

Scope Singleton чаще всего используется тогда, когда у нашего бина нет изменяемых состояний (stateless). Потому что если потом будем изменять состояние у Singleton бина, столкнемся с проблемой.

Можно явно указывать scope Singleton.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | <bean id="musicBean"  class="springFramework.ClassicalMusic"  scope="singleton">  </bean> |

Этот паттерн программирования надо знать

Ссылка на репозиторий с Java‑кодом (и комментариями) будет в описании

Prototype

Scope, который каждый раз создает новый объект при вызове getBean().

|  |  |
| --- | --- |
| 1.  2.  3. | MusicPlayer firstMusicPlayer  = context.getBean("musicPlayer",  MusicPlayer.class); |
| 1.  2.  3. | MusicPlayer secondMusicPlayer  = context.getBean("musicPlayer",  MusicPlayer.class); |

На каждую переменную новый объект

Scope Prototype чаще всего используется тогда, когда у нашего бина есть изменяемые состояния (stateful).

Другие scope

singleton, prototype, request, session, global‑session

Урок 8: Жизненный цикл бина. Init, Destroy и Factory методы.

Способы внедрения зависимостей

Через конструктор

Через setter

Можем внедрять ссылки или простые значения

Можем внедрять значения из внешнего файла

У бинов есть множество конфигураций (scope, init‑method, destroy‑method, factory method и т.д.)

Можно внедрять через XML, аннотации или Java‑код

Процесс внедрения можно автоматизировать (Autowiring)

Жизненный цикл бина (Bean Lifecycle)

|  |  |
| --- | --- |
| 1.  2.  3. | <bean id="musicBean"  class="springFramework.ClassicalMusic">  </bean> |

1. вы запускаете Spring приложение
2. запускается Spring контейнер
3. создается объекты бина
4. в бин внедряются зависимости
5. вызывается указанный init‑method
6. бин готов к использованию
7. бин передан пользователю
8. вызывается указанный destroy‑method
9. остановка Spring приложения

init и destoy методы

init‑method

Метод, который запускается в ходе инициализации бина.

Инициализация ресурсов, обращение к внешним файлам, запуск БД.

destoy‑method

Метод, который запускается в ходе уничтожения бина (при завершении приложения).

Очищение ресурсов, закрытие потоков ввода‑вывода, закрытие доступа к БД.

В коде

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | <bean id="musicBean"  class="springFramework.ClassicalMusic"  init-method="doMyInit"  destroy-method="doMyDestroy">  </bean> |

Методы doMyInit() и doMyDestroy() создаются в классе бина.

Тонкости init и destroy методов

Модификатор доступа.

У этих методов может быть любой модификатор доступа (public, protected, private)

Тип возвращаемого значения.

Может быть любой, но чаще всего используется void (так как нет возможности получить возвращаемое значение)

Название метода.

Название метода может быть любым

Аргументы метода.

Эти методы не должны принимать на вход какие‑либо аргументы

Еще тонкости.

Для банов со csope "prototype" Spring не вызывает destroy метод

Spring не берет на себя полный жизненный цикл бинов со scope "prototype". Spring отдает prototype бины клиенту и больше о них не заботиться (в отличии от singleton бинов).

factory‑method

Фабричный метод (англ. Factory Method) — это паттерн программирования

Вкратце: паттерн «фабричный метод» предлагает создавать объекты не напрямую, используя операторы new, а через вызов особого фабричного метода. Объекты все равно будут создаваться при помощи new, но делать это будет фабричный метод (иногда это бывает полезно).

Очень хорошее объяснение: https://refactoring.guru/ru/design-patterns/factory-method

Если объекты класса создаются фабричным методом, то можно определить factory-method.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | <bean id="musicBean"  class="springFramework.ClassicalMusic"  factory-method="getClassicalMusic">  </bean> |

Урок 9: Аннотации. Введение

Что такое Java Аннотации?

Java Аннотации — это специальный метод комментариев в вашем коде с помощью которых можно:

1. представить какие‑либо инструкции для Java компилятора (@Override)
2. передавать какие‑либо инструкции для анализаторов исходного кода
3. передавать метаданные, которые могут быть использованы либо вашим Java приложением (с помощью рефлексии), либо другими приложениями или фреймворками (Spring Framework)

Зачем использовать аннотации?

Короче, чем XML конфигурация

Удобнее, чем XML конфигурация

Код становится более читабельным

Как работает конфигурация с помощью аннотаций?

Spring сканирует все ваши классы

Находит классы со специальными аннотациями и автоматически создает бины их этих классов

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | <bean id="musicBean"  class="springFramework.ClassicalMusic"  init-method="doMyInit"  destroy-method="doMyDestroy">  </bean> |
| 1.  2.  3.  4.  5.  6.  7.  8. | @Component  public class ClassicalMusic  implenents Music {  @Override  public String getSong() {  return "классическая музыка";  }  } |

Аннотация @Component

Помечаем ей классы, если хотим, чтобы Spring Framework создал бин из этого класса

Именно эту аннотацию Spring Framework ищет, когда сканирует все ваши классы

Можно указать id для создаваемого бина, можно не указывать (тогда название будет название\_класса\_с\_маленькой\_буквы).

Что мы сделаем на этом уроке?

Уберем лишний XML с предыдущих уроков (создание бинов)

Включим сканирование компонентов в конфигурационном файле Spring

Добавим аннотацию @Component к нашим Java классам

Получим созданный бин из Application Context

Урок 10: Аннотация @Autowired. DI

На прошлом уроке

Создали два бина с помощью Spring аннотации

На этом уроке мы начнем внедрять зависимости с помощью аннотации

Аннотация @Autowired

Мы больше не внедряем зависимость вручную, Spring сам ищет подходящий бин и автоматически внедряет его.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | <bean id="musicBean"  class="springFramework.ClassicalMusic">  </bean>  <bean id="musicPlayer"  class="springFramework.MusicPlayer">  <constructor-arg ref="musicBean"/>  </bean> |
| 1.  2.  3.  4. | @Autowired  public MusicPlayer(Music music) {  this.music = music;  } |

Как работает аннотация @Autowired?

В данном примере в бин musicPlayer необходимо внедрить бин, который реализует интерфейс Music

|  |  |
| --- | --- |
| 1.  2.  3.  4. | @Autowired  public MusicPlayer(Music music) {  this.music = music;  } |

Spring сканирует все классы с аннотацией @Component и создает бины для этих классов

Spring сканирует все созданные бины и проверяет, подходит ли хотя бы один бин в качестве зависимости там, где мы указали аннотацию @Autowired

Если находится один подходящий бин, он внедряется в качестве зависимости

Если не находится ни одного бина — ошибка

Если несколько бинов подходят — неоднозначность (будет рассмотрена на следующем уроке).

Еще об @Autowired

|  |  |
| --- | --- |
| 1.  2.  3.  4. | @Autowired  public MusicPlayer(Music music) {  this.music = music;  } |
| 1.  2.  3.  4. | @Autowired  public MusicPlayer(Music classicalMusic) {  this.classicalMusic = classicalMusic;  } |

Аннотация @Autowired подирает подходящие бины по их типу (класс или интерфейс)

|  |  |
| --- | --- |
| 1.  2. | @Autowired  private Music music; |
| 1.  2.  3.  4. | @Autowired  public void setMusic(Music music) {  this.music = music;  } |
| 1.  2.  3.  4. | @Autowired  public MusicPlayer(Music music) {  this.music = music;  } |

Аннотацию @Autowired можно использовать на полях, сеттерах и конструкторах.

|  |  |
| --- | --- |
| 1.  2. | @Autowired  private Music music; |

Аннотация @Autowired внедрит зависимость в приватное поле даже если нет конструктора или сеттера. Делает она это с помощью Рефлексии (Java Reflection API).

Что мы сделаем на этой практике?

Начнем работу с прокта, который остался с предыдущего урока

Внедрим зависимость через конструкторы, сеттеры и поля

Разберем некоторые тонкости аннотации @Autowired (приватные поля и т.д.)

Увидим ошибку, когда Spring не находит подходящего бина для внедрения

Увидим проблему неоднозначности, когда несколько бинов подходят для внедрения

P.S.

На практике мы изучили три способа внедрения зависимостей — через конструктор, через сеттеры, напрямую через поле.

Какой из способов использовать?

Выберете один из способов (любой, так как они одинаковы по функциональности) и используйте его повсеместно в вашем проекте. Главное — это стараться не смешивать разные стили, а придерживаться выбранного.

Урок 11: Аннотация @Qualifier. DI

Проблема

Необходимо обозначить, какой из бинов хотим внедрить

Аннотация @Qualifier

англ. «Уточнитель»

|  |  |
| --- | --- |
| 1.  2.  3. | @Autowired  @Qualifier("rockMusic")  private Music music; |

Указываем id того бина, который хотим внедрить

Аннотацию можно использовать на конструкторах, сеттерах, полях.

Необычный синтаксис

Когда внедрение производится с помощью конструктора, для уточнения зависимостей необходимо использовать такой синтаксис:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | @Autowired  public MusicPlayer(  @Qualifier("rockMusic") Music music1,  @Qualifier("classicalMusic")  Music music2) {  this.music1 = music1;  this.music2 = music2;  } |

Аннотацию @Qualifier необходимо использовать рядом с аргументами конструктора.

Самостоятельное задание

Создайте список (или массив) из 3 песен в ClassicalMusic и RockMusic (по 3 песни на каждый жанр)

В MusicPlayer должны внедряться бины ClassicalMusic и RockMusic

Создать Enum с двумя жанрами музыки — CLASSICAL и ROCK

Переделать метод playMusic() так, чтобы он принимал на вход этот Enum

В зависимости от значения этого Enum (CLASSICAL или ROCK) метод playMusic() должен воспроизвести случайную классическую или рок песню (использовать класс Random)

Урок 12: Аннотации @Scope, @Value, @PostConstruct, @PreDestroy

Внедрение значения из внешнего файла

Как мы делали с помощью XML—конфигурации

1. musicPlayer.properties

|  |  |
| --- | --- |
| 1.  2. | musicPlayer.name=Some name  musicPlayer.volume=60 |

|  |  |
| --- | --- |
| 1.  2. | <context:property-placeholder location="  classpath:musicPlayer.properties"/> |

|  |  |
| --- | --- |
| 1.  2.  3.  4. | <property name="name"  value="${musicPlayer.name}"/>  <property name="volume"  value="${musicPlayer.volume}"/> |

Аннотации @Value

1. musicPlayer.properties

|  |  |
| --- | --- |
| 1.  2. | musicPlayer.name=Some name  musicPlayer.volume=60 |

|  |  |
| --- | --- |
| 1.  2. | <context:property-placeholder location="  classpath:musicPlayer.properties"/> |

|  |  |
| --- | --- |
| 1.  2.  3.  4. | @Value("${musicPlayer.name}")  private String name;  @Value("${musicPlayer.volume}")  private int volume; |

Scope

«Область видимости» бинов

|  |  |
| --- | --- |
| 1.  2.  3.  4. | <bean id="musicBean"  class="springFramework.ClassicalMusic"  scope="singleton">  </bean> |

Scope задает то, как Spring будет создавать ваши бины (по умолчанию — Singleton)

Отличие Singleton от Prototype

При Scope Singleton при каждом вызове метода getBean() возвращается один и тот же объект их Spring Application Context. При Scope Prototype при каждом вызове getBean() будет создаваться новый объект.

Аннотация @Scope

Singleton

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | @Component  @Scope("singleton")  public class ClassicalMusic  implenents Music {  @Override  public String getSong() {  return "классическая музыка";  }  } |

Prototype

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | @Component  @Scope("prototype")  public class ClassicalMusic  implenents Music {  @Override  public String getSong() {  return "классическая музыка";  }  } |

Жизненный цикл бина (Bean Lifecycle)

1. вы запускаете Spring приложение
2. запускается Spring контейнер
3. создается объекты бина
4. в бин внедряются зависимости
5. вызывается указанный init‑method
6. бин готов к использованию
7. бин передан пользователю
8. вызывается указанный destroy‑method
9. остановка Spring приложения

init и destoy методы

init‑method

Метод, который запускается в ходе инициализации бина.

Инициализация ресурсов, обращение к внешним файлам, запуск БД.

destoy‑method

Метод, который запускается в ходе уничтожения бина (при завершении приложения).

Очищение ресурсов, закрытие потоков ввода‑вывода, закрытие доступа к БД.

Аннотации @PostConstruct и @PreDestroy

...

1. вызывается указанный init‑method/@PostConstructor
2. бин готов к использованию
3. бин передан пользователю
4. вызывается указанный destroy‑method/@PreDestroy

...

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | @Component  public class ClassicalMusic  implenents Music {  @PostConstruct  public void doMyInit() {  System.out.println(  "Инициализация бина");  }  @PreDestroy  public void doMyDestroy() {  System.out.println(  "Уничтожение бина");  }  } |

Сигнатура @PostConstruct и @PreDestroy методов

Модификатор доступа

У этих методов может быть любой модификатор доступа (public, protected, private)

Тип возвращаемого значения

Может быть любой, но чаще всего используется void (так как нет возможности поулчить возвращаемое значение)

Название метода

Название метода может быть любым

Аргумент метода

Эти методы не должны принимать на вход какие‑либо аргументы

Урок 13: Конфигурация с помощью Java кода

Конфигурация Spring приложений

1. Вся конфигурация на XML

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29. | <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org  /schema/beans"  xmlns:xsi="http://www.w3.org/2001  /XMLSchema-instance"  xmlns:context="  http://www.springframework.org  /schema/context"  xsi:schemaLocation="  http://www.springframework.org  /schema/beans  http://www.springframework.org  /schema/beans/spring-beans.xsd  http://www.springframework.org  /schema/context  http://www.springframework.org  /schema/context/spring-context.xsd">  <bean id="musicBean"  class="springFramework.RockMusic">  </bean>  <bean id="musicPlayer"  class="  springFramework.MusicPlayer">  <constructor-arg ref="musicBean"/>  </bean>  </beans> |

1. XML и Аннотации

|  |  |
| --- | --- |
| 1.  2. | <context:component-scan  base-package="springFramework"/> |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | @Component  public class MusicPlayer {  private Music music1;  private Music music2;  @Autowired  public MusicPlayer(  @Qualifier("classicalMusic")  Music music1,  @Qualifier("rockMusic")  Music music2) {  this.music1 = music1;  this.music2 = music2;  }  } |

1. Java код и аннотации

XML конфигурации больше нет

Аннотация @Configuration

Помечаем Java класс, который мы хотим использовать для конфигурации Spring приложения

|  |  |
| --- | --- |
| 1.  2.  3. | @Configuration  public class SpringConfig {  } |

Пустой конфигурационный Java класс равен по функционалу пустому конфигурационному XML файлу

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org  /schema/beans"  xmlns:xsi="http://www.w3.org/2001  /XMLSchema-instance"  xmlns:context="  http://www.springframework.org  /schema/context"  xsi:schemaLocation="  http://www.springframework.org  /schema/beans  http://www.springframework.org  /schema/beans/spring-beans.xsd  http://www.springframework.org  /schema/context  http://www.springframework.org  /schema/context/spring-context.xsd">  </beans> |

Для каждого XML тега есть соответствующая аннотация

|  |  |
| --- | --- |
| 1.  2. | <context:component-scan  base-package="springFramework"/> |
| 1.  2.  3.  4. | @Configuration  @ComponentScan("springFramework")  public class SpringConfig {  } |

|  |  |
| --- | --- |
| 1.  2.  3. | <bean id="musicBean"  class="springFramework.RockMusic">  </bean> |
| 1.  2.  3.  4.  5.  6.  7. | @Configuration  public class SpringConfig {  @Bean  public ClassicalMusic musicBean() {  return new ClassicalMusic();  }  } |

И так далее для всех тегов в XML конфигурации

Использование конфигурационного файла

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | public static void main(String[] args) {  ClassPathXmlApplicationContext context  = new  ClassPathXmlApplicarionContext(  "applicationContext.xml"  );  MusicPlayer musicPlayer  = context.getBean("musicPlayer",  MucisPlayer.class);  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | public static void main(String[] args) {  AnnotationConfigApplicationContext  Context  = new AnnotationConfig  ApplicationContext(  SpringConfig.class);  MusicPlayer musicPlayer  = context.getBean("musicPlayer",  MucisPlayer.class);  } |

Ручное внедрение зависимостей (без @Autowired) с помощью Java конфигурации

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | <bean id="musicBean"  class="springFramework.RockMusic">  </bean>  <bean id="musicPlayer"  class="springFramework.MusicPlayer">  <constructor-arg ref="musicBean"/>  </bean> |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | @Configuration  public class SpringConfig {  @Bean  public ClassicalMusic musicBean() {  return new ClassicalMusic();  }  @Bean  public MusicPlayer musicPlayer() {  return new MusicPlayer(musicBean());  }  } |

Побробнее об аннотации @Bean

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | @Configuration  public class SpringConfig {  @Bean  public ClassicalMusic musicBean() {  return new ClassicalMusic();  }  @Bean  public MusicPlayer musicPlayer() {  return new MusicPlayer(musicBean());  }  } |

Как и в случае с XML конфигурацией, по умолчанию у бинов scope=singleton

Это значит, что тело @Bean методов по умолчанию вызывается только один раз, в все последующие вызовы Spring прерывает и возвращает уже имеющийся бин из контекста

Внедрение значений из внешнего файла

Раньше

1. musicPlayer.properties

|  |  |
| --- | --- |
| 1.  2. | musicPlayer.name=Some name  musicPlayer.volume=60 |

1. applicationContext.xml

|  |  |
| --- | --- |
| 1.  2. | <context:property-placeholder location="  classpath:musicPlayer.properties"/> |

1. MusicPlayer.java

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @Value("${musicPlayer.name}")  private String name;  @Value("${musicPlayer.volume}")  private int volume; |

Теперь

1. musicPlayer.properties

|  |  |
| --- | --- |
| 1.  2. | musicPlayer.name=Some name  musicPlayer.volume=60 |

1. SpringConfig.java

applicationContext.xml

|  |  |
| --- | --- |
| 1.  2. | @Configuration  @PropertySource(  "classpath:musicPlayer.properties")  public class SpringConfig {  } |

1. MusicPlayer.java

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @Value("${musicPlayer.name}")  private String name;  @Value("${musicPlayer.volume}")  private int volume; |

Самостоятельное задание

Создайте еще один класс — жанр музыки

Измените MusicPlayer так, чтобы он хранил в себе не отдельные поля для музыкальных жанров, а хранил в себе список жанров (List)

Поменяйте конструктор соответственно, чтобы он ожидал на вход список из музыкальных жанров

Создайте бин для нового жанра музыки в конфигурационном файле

Создайте бин, который представляет из себя список из музыкальных жанров

Внедрите этот бин‑список в бин MusicPlayer

Измените метод playMusic() так, чтобы он воспроизводил песню случайного музыкального жанра (класс Random)

Аннотацию @Component, @Autowired и @ComponentScan не использовать — все сделать вручную в конфигурационном файле

Урок 14: Spring MVC. Теория

Где мы находимся?

Прошли: Spring Core — IoC, DI, Beans, Configuration (XML, Annitation и Java), Autowiring

Мы здесь: Spring MVC

...

Hibenate, Spring Boot, Spring Security, etc

Что такое Spring MVC

Один из компонентов Spring Framework, который позволяет разрабатывать web‑приложения на Java

Spring MVC предполагает разработку web‑приложений с использованием архитектуры Model‑View‑Controller

Разрабатывая web‑приложения с помощью Spring MVC, мы можем использовать все, что дает нам Spring Core — бины, DI и так далее

MVC (Model‑View‑Controller)

MVC — паттерн проектирования приложения

Model — логика работы с данными

View — логика представления, интерфейс

Controller — логика навигации, обработка запросов

Из чего состоит Spring MVC приложение

Из обычных Java классов (контроллеры, модели и прочее). Очень активно используются аннотации, которые применяются к классам и дают им дополнительные возможности (например, аннотация @Controller)

Набор HTML страниц (представления). К ним часто добавляется JavaScript код, который «оживляет» HTML представления и CSS, который стилизует HTML

String конфигурация (XML, Аннотации, или Java)

DispatcherServet

Является входной точкой Spring MVC приложения

Реализован за нас командой Spring

Мы реализуем Модели, Представления и Контроллеры (MVC)

HTTP запрос от пользователя

Приходит на сервер. Сервер обрабатывает запрос и передает его на Spring MVC приложение

Запрос попадает в DispatcherServet

DispatcherServet отправляет запрос на правильный контроллер

Контроллер (Controller)

Контроллер реализуется нами

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | ... |

Обрабатывает запросы от пользователя

Обменивается данными с моделью

Показывает пользователю правильное представление

Переадресовывает пользователя на другие страницы

Модель (Model)

Модель реализуется нами

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | ... |

Хранит в себе данные

Взаимодействует с БД для получения данных

Отдает данные контроллеру

Представление (View)

Представление реализуется нами

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | ... |

Получает данные от контроллера и отображает их в браузере

Для динамического отображения данных используются шаблонизаторы (Thymeleaf, Freemarker, Velocity)

На следующем уроке

Реализуем наше первое Spring MVC приложение

Урок 15. Spring MVC. Первое приложение

Что мы сделаем на этом уроке?

Создадим новое пустое Spring приложение (как в одном из первых уроков)

Подключим Tomcat к IntelliJ Idea (чтобы удобно запускать сервер прямо из IDE)

Добавим необходимые зависимости в pom.xml

Реализуем web.xml

Реализуем applicationContext.xml

Реализуем контроллер и преставление (пока без модели)