Инверсия управления (IoC) — принцип объектно-ориентированного программирования, используемый для уменьшения связанности кода.

В Java — программные на уровне интерфейсов.

DI — реализация этого принципа (фактори)

WAR — формат файла, описывающий, как полное веб-приложение упаковывается в соответствии со спецификацией Java-сервлетов в файл в формате JAR или ZIP.[2] Такие файлы имеют расширение «.war» и поэтому называются ещё «WAR-файлами».

Скопе:

singleton

единственный бин

prototype

любое количество экземпляров бина

request

один экземпляр на запрос

session

один экземпляр на сессию.

global-session

один экземпляр на глобальную сессию

В сервис слое сидит бизнес логика, для реализации которой он обращается к разным DAO.

@Service - мы используем данную аннотацию, чтобы объявить, что этот класс представляет сервис – компонент сервис-слоя. Сервис является подтипом класса @Component. Использование данной аннотации позволит искать бины-сервисы автоматически.

@Transactional

Жизненный цикл бина

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

Способы внедрения зависимостей

Через конструктор или setter; ссылки или простые значения; значения из внешнего файла; конфигурации (scope, init‑method, destroy‑method, factory method и т.д.); 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)

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.

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

@Configuration — класс, который мы хотим использовать для конфигурации Spring приложения

@ComponentScan("") — указываются пакеты

Ручное внедрение зависимостей (без @Autowired)

|  |  |
| --- | --- |
| 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());  }  } |

@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());  }  } |

scope=singleton

Spring MVC

Компонент Spring Framework, позволяющий разрабатывать web‑приложения

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

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)

Обрабатывает запросы от пользователя

Обменивается данными с моделью

Показывает пользователю правильное представление

Переадресовывает пользователя на другие страницы

Модель (Model)

Хранит в себе данные (взаимодействие с БД, передача данные контроллеру)

Представление (View)

Получает данные от контроллера и отображает их в браузере