Spring

Автор:
Эльдар Суйналиев

При участии:
Антон Таврель
Роман Евсеев

2020 год

Оглавление

Введение

В настоящем материале используется официальная документация и руководства Spring Framework. Практические примеры преимущественно получены с сайтов, пользующихся популярностью и уважением в мировом сообществе разработчиков: Baeldung и другие.

Использованные версии документаций:
Spring Framework Documentation Version 5.2.7.RELEASE Reference Doc.
Spring Framework 5.2.7.RELEASE API

Если Вы заметите неточность или ошибку, либо захотите дополнить ответ - пиши мне, и мы всё обсудим.

1. Что такое инверсия контроля (IoC) и внедрение зависимостей (DI)? Как они реализованы в Spring?

Источники: Spring - Introduction to the Spring IoC Container and Beans
Baeldung - Inversion of Control and Dependency Injection
Martin Fowler - Inversion Of Control

Inversion of Control (IoC)

Инверсия (от латинского inversio) - перестановка.

Инверсия контроля (инверсия управления) — это принцип в разработке программного обеспечения, при котором управление объектами или частями программы передается контейнеру или фреймворку. Чаще всего этот принцип используется в контексте объектно-ориентированного программирования.

В отличие от традиционного программирования, в котором наш пользовательский код обращается напрямую к библиотекам, IoC позволяет фреймворку контролировать ход программы и обращаться к нашему коду, когда это необходимо. Для этого, фреймворки используют абстракции со встроенным дополнительным поведением. Если мы хотим добавить наше собственное поведение, нам нужно расширить классы фреймворка или подключить наши собственные классы.

Преимущества этой архитектуры:

- отделение выполнения задачи от ее реализации;
- легкое переключение между различными реализациями;
- большая модульность программы;
- более легкое тестирование программы путем изоляции компонента или проверки его зависимостей и обеспечения взаимодействия компонентов через контракты.

Инверсия управления может быть достигнута с помощью различных механизмов, таких как: шаблон проектирования “Стратегия”, шаблон “Локатор служб”, шаблон “Фабрика” и внедрение зависимостей (DI).

Dependency Injection (DI)

Внедрение зависимостей — это шаблон проектирования для реализации IoC, где инвертируемым (переопределяемым) элементом контроля является настройка зависимостей объекта.

Соединение объектов с другими объектами или «внедрение» объектов в другие объекты выполняется контейнером IoC, а не самими объектами.

В Spring Framework инверсия контроля достигается именно внедрением зависимостей. В Spring Framework инверсия контроля и внедрение зависимостей считаются одним и тем же.

В Spring Framework внедрение зависимостей описывается как процесс, посредством которого объекты определяют свои зависимости (то есть другие объекты, с которыми они работают) только через аргументы конструктора, аргументы фабричного метода[1] или свойства,

[1] Шаблон проектирования “Фабричный метод” предлагает создавать объекты не напрямую, используя оператор **new**, а через вызов особого статического фабричного метода в классе. Объекты всё равно будут создаваться через оператор **new**, но делать это будет фабричный метод. Refactoring Guru - Фабричный метод

которые устанавливаются в экземпляре объекта после того, как он создан или возвращен из метода фабрики. После чего контейнер IoC внедряет эти зависимости в компонент при его создании.

Мы можем создать зависимость объекта следующим традиционным способом, без использования принципа IoC:

```
public class Store {
    private Item item;
    public Store() {
        item = new ItemImpl1();
    }
}
```

В приведенном выше примере мы создаем экземпляр конкретной реализации интерфейса Item (ItemImpl1) внутри самого класса Store.

Используя DI, мы можем переписать пример без указания конкретной реализации Item, не создавая её внутри нашего объекта, а ожидая её получение извне (от внешнего фреймворка - контейнера IoC):

```
public class Store {
    private Item item;
    public Store(Item item) {
        this.item = item;
    }
}
```

В данном случае **инверсия контроля** — это переход контроля над зависимостями от объекта Store к контейнеру IoC. Объект Store более не контролирует инстанцирование своего поля (зависимости) item, не создаёт этот объект самостоятельно, а делегирует этот процесс внешним силам - контейнеру IoC, который в нашем примере передаёт в конструктор Store любую из реализаций Item.

Внедрение зависимостей в Spring Framework может быть сделано через конструкторы, сеттеры или поля.

Отдаем создание и управление объектами на аутсорс Spring-у.

2. Что такое IoC Container?

Источники: Spring - The IoC Container
Baeldung - The Spring IoC Container

IoC Container

В Spring Framework контейнер отвечает за создание, настройку и сборку объектов, известных как **бины**, а также за управление их жизненным циклом. Он (контейнер) представлен интерфейсом **ApplicationContext**.

Spring Framework предоставляет несколько реализаций интерфейса ApplicationContext:

- ClassPathXmlApplicationContext и FileSystemXmlApplicationContext - для автономных приложений;
- WebApplicationContext - для веб-приложений;
- AnnotationConfigApplicationContext - для обычной Java-конфигурации, в качестве аргумента которому передается класс, либо список классов с аннотацией @Configuration, либо с любой другой аннотацией JSR-330, в том числе и @Component.

Контейнер получает инструкции о том, какие объекты создавать, настраивать и собирать, через **метаданные конфигурации**, которые представлены в виде XML, Java-аннотаций или Java-кода:

- XML - Метаданные считываются из файла с расширением *.xml;
- Java-аннотации - В Spring 2.5 появилась поддержка метаданных конфигурации на основе аннотаций, которая использует данные байт-кода для подключения компонентов. Вместо того, чтобы использовать XML-файл для описания связывания компонентов, разработчик перемещает конфигурацию в сам класс компонента, используя аннотации к соответствующему классу, методу или полю. При этом, сам XML-файл с базовыми настройками остаётся. Контейнер считывает аннотации перед считыванием XML, поэтому, если бин конфигурируется и через аннотации и через XML-файл, то настройки XML переопределят настройки аннотаций.
- Java-код - Начиная со Spring 3.0, используя Java-код, а не файлы XML, мы можем определять настройки в специальном классе, помеченном аннотацией @Configuration. Появились аннотации @Configuration, @Bean, @Import и @DependsOn и т.д.

3. Что такое Bean в Spring?

Источники: Spring - The IoC Container

В Spring объекты, образующие основу приложения и управляемые контейнером Spring IoC, называются **бинами**.

Бин — это объект, который создается, собирается и управляется контейнером Spring IoC. Иначе говоря, бин — это просто один из множества объектов в вашем приложении. Бины и их зависимости отражаются в метаданных конфигурации, используемых контейнером.

4. Расскажите про аннотацию @Bean

Источники:
Spring - The IoC Container
Spring - How Java-based Configuration Works Internally
Spring API - @Bean
Stackoverflow - @Configuration vs. @Component
LogicBig - Registering beans within @Component classes

@Bean - Это аннотация Spring Framework, она используется над методом для указания того, что данный метод создает, настраивает и инициализирует новый объект, управляемый Spring IoC контейнером. Такие методы можно использовать как в классах с аннотацией @Configuration, так и в классах с аннотацией @Component (или её наследниках).

Позволяет дополнительно определить у бина:

- name - имя (уникальный идентификатор) бина;
- initMethod - имя метода для вызова во время инициализации бина;
- destroyMethod - имя метода для вызова во время удаления бина из контекста;
- autowireCandidate - является ли этот бин кандидатом на автоматическое внедрение в другой бин.

Классы, аннотированные @Configuration, проксируются через CGLIB. Классы @Component или обычные классы не проксируются и не перехватывают вызовы методов с аннотациями @Bean, что означает, что вызовы не будут маршрутизироваться через контейнер и каждый раз будет возвращаться новый экземпляр бина.

Также методы бинов, вызывая друг друга в таких классах, не будут создавать бины, а будет просто выполняться код метода, ведь в данном случае они отработают не через прокси.

CGLIB (Code Generation Library) - Это библиотека инструментария байтов, используемая во многих средах Java, таких как Hibernate или Spring. Инструментарий байт-кода позволяет манипулировать или создавать классы после фазы компиляции программы.

Hibernate использует cglib для генерации динамических прокси. Например, он не вернет полный объект, хранящийся в базе данных, но вернет инструментальную версию хранимого класса, которая лениво загружает значения из базы данных по требованию.

Прокси — это шаблон проектирования. Создаем и используем его для добавления и изменения функционала уже существующих классов. В таком случае, прокси-объект применяется вместо исходного. Обычно он использует тот же метод, что и оригинальный, и в Java прокси-классы расширяют исходные.

Имена бинов

Имя бина, которое в контейнере является одновременно и его уникальным идентификатором, по умолчанию соответствует имени метода, аннотированного @Bean. Но если требуется указать иное имя, то можно использовать атрибут name, который принимает String. Однако, атрибут name также может принимать массив String, что позволяет использовать несколько имен. Первый элемент массива будет являться именем и уникальным идентификатором бина, а остальные будут его псевдонимами.

```
@Bean({"b1", "b2"}) // bean available as 'b1' and 'b2', but not 'myBean'
```

```
public MyBean myBean() {
    // instantiate and configure MyBean obj
    return obj;
}
```

5. Расскажите про аннотацию @Component

Источники: Spring - @Component and Further Stereotype Annotations
Shell26 - Различия @Component, @Service, @Repository, @Controller

Это аннотация Spring Framework, ею мы помечаем класс, если хотим, чтобы из этого класса был создан бин. Именно эту аннотацию ищет Spring Framework, когда сканирует наши классы. Можно указать имя (Id) для создаваемого бина, а можно не указывать, тогда по умолчанию именем будет название класса с маленькой буквы.

Аннотация @Component имеет наследников: @Repository, @Service и @Controller. Все они являются частными случаями использования @Component для слоёв DAO, сервиса и контроллера MVC соответственно. Также эти аннотации могут иметь дополнительный смысл в будущих версиях Spring Framework. В остальных же случаях достаточно использовать аннотацию @Component.

Итог:

- @Component - Spring определяет этот класс как кандидата для создания bean.
- @Service - класс содержит бизнес-логику и вызывает методы на уровне хранилища. Ничем не отличается от классов с @Component.
- @Repository - указывает, что класс выполняет роль хранилища (объект доступа к DAO). Задача @Repository заключается в том, чтобы отлавливать определенные исключения персистентности и пробрасывать их как одно непроверенное исключение Spring Framework. Для этого Spring оборачивает эти классы в прокси, и в контекст должен быть добавлен класс PersistenceExceptionTranslationPostProcessor.
- @Controller - указывает, что класс выполняет роль контроллера MVC. DispatcherServlet просматривает такие классы для поиска @RequestMapping.

@RequestMapping используется для мапинга (связывания) с URL для всего класса или для конкретного метода обработчика.

6. Чем отличаются аннотации @Bean и @Component?

Источники: Stackoverflow - @Component versus @Bean

Аннотация @Component (как и @Service и @Repository) используется для автоматического обнаружения и автоматической настройки бинов в ходе сканирования путей к классам.

Аннотация @Bean используется для явного объявления бина, а не для того, чтобы Spring делал это автоматически в ходе сканирования путей к классам:

- прописываем вручную метод для создания бина;
- делает возможным объявление бина независимо от объявления класса, что позволяет использовать классы из сторонних библиотек, у которых мы не можем указать аннотацию @Component;
- с аннотацией @Bean можно настроить initMethod, destroyMethod, autowireCandidate, делая создание бина более гибким.

7. Расскажите про аннотации @Service и @Repository. В чем различия?

Источники:
Spring - The IoC Container
Baeldung - @Component vs @Repository and @Service
Baeldung - Spring Bean Annotations

@Service и @Repository являются частными случаями @Component. Технически они одинаковы, но мы используем их для разных целей.

Задача @Repository заключается в том, чтобы отлавливать определенные исключения персистентности и пробрасывать их как одно непроверенное исключение Spring Framework. Для этого в контекст должен быть добавлен класс PersistenceExceptionTranslationPostProcessor.

Мы помечаем бины аннотацией @Service, чтобы указать, что они содержат бизнес-логику. Так что нет никакого другого предназначения, кроме как использовать ее на уровне сервиса.

8. Расскажите про аннотацию @Autowired

Источники:
Spring - Using @Autowired
Spring API - @Autowired
Baeldung - Guide to Spring @Autowired
Baeldung - @Qualifier vs Autowiring by Name
Shell26 - @Autowired vs @Resource vs @Inject

Это аннотация Spring Framework, ею помечают конструктор, поле, сеттер-метод или метод конфигурации, сигнализируя, что им обязательно требуется внедрение зависимостей.

Если в контейнере не будет обнаружен необходимый для вставки бин, то будет выброшено исключение, либо можно указать @Autowired(required = false), означающее, что внедрение зависимости в данном месте не обязательно.

Аннотация @Autowired является альтернативой Java-аннотации @Inject, не имеющей required = false (зависимость должна быть обязательно внедрена).

Начиная со Spring Framework 4.3, аннотация @Autowired для конструктора больше не требуется, если целевой компонент определяет только один конструктор. Однако, если доступно несколько конструкторов и нет основного/стандартного конструктора, по крайней мере один из конструкторов должен быть аннотирован @Autowired, чтобы указать контейнеру, какой из них использовать.

По умолчанию Spring распознает объекты для вставки по типу. Если в контейнере доступно более одного бина одного и того же типа, будет исключение. Для избежания этого можно указать аннотацию Spring Framework - @Qualifier("fooFormatter"), где fooFormatter — это имя (Id) одного из нескольких бинов одного типа, находящихся в контейнере и доступных для внедрения:

```
public class FooService {
    @Autowired
    @Qualifier("fooFormatter")
    private Formatter formatter;
}
```

При выборе между несколькими бинами при автоматическом внедрении используется **имя поля**. Это поведение по умолчанию, если нет других настроек. Давайте посмотрим код, основанный на нашем первоначальном примере:

```
@Component
@Qualifier("fooFormatter")
public class FooFormatter implements Formatter {
    //...
}


@Component
@Qualifier("barFormatter")
public class BarFormatter implements Formatter {
    //...
}
```

```
public class FooService {
    @Autowired
    private Formatter fooFormatter;
}
```

В этом случае Spring определит, что нужно внедрить бин с именем FooFormatter, поскольку имя поля соответствует значению, которое мы использовали в аннотации @Component для этого бина.

Мы также можем указать Spring предоставить **все** бины определенного типа из ApplicationContext, добавив аннотацию @Autowired в поле или метод с массивом или коллекцией этого типа, как показано в следующем примере:

```
@Autowired
private MovieCatalog[] movieCatalogs;
```

или:

```
@Autowired
private Set<MovieCatalog> movieCatalogs;
```

или:

```
@Autowired
public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {
    this.movieCatalogs = movieCatalogs;
}
```

Даже коллекции типа Map могут быть подключены автоматически, если тип ключа - String. Ключами будут имена бинов, а значениями - сами бины, как показано в следующем примере:

```
public class MovieRecommender {

   private Map<String, MovieCatalog> movieCatalogs;

   @Autowired
   public void setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs){
      this.movieCatalogs = movieCatalogs;
   }
   // ...
}
```

9. Расскажите про аннотацию @Resource

Источники:
- Javadoc - @Resource
- Baeldung - The @Resource Annotation
- Stackoverflow - @Resource vs @Autowired
- Shell26 - @Autowired vs @Resource vs @Inject

Java-аннотация @Resource может применяться к классам, полям и методам. Она пытается получить зависимость: сначала по имени, затем по типу, затем по описанию (Qualifier). Имя извлекается из имени аннотируемого сеттера или поля, либо берется из параметра name. При аннотировании классов имя не извлекается из имени класса по умолчанию, поэтому оно должно быть указано явно.

Указав данную аннотацию у полей или методов с аргументом name, в контейнере будет произведен поиск компонентов с данным именем, и в контейнере должен быть бин с таким именем:

```
@Resource(name="namedFile")
private File defaultFile;
```

Если указать её без аргументов, то Spring Framework поможет найти бин по типу.

Если в контейнере несколько бинов-кандидатов на внедрение, то нужно использовать аннотацию @Qualifier:

```
@Resource
@Qualifier("defaultFile")
private File dependency1;

@Resource
@Qualifier("namedFile")
private File dependency2;
```

Разница с @Autowired:

- ищет бин сначала по имени, а потом по типу;
- не нужна дополнительная аннотация для указания имени конкретного бина;
- @Autowired позволяет отметить место вставки бина как необязательное @Autowired(required = false);
- при замене Spring Framework на другой фреймворк, менять аннотацию @Resource не нужно.

10. Расскажите про аннотацию @Inject

Источники:
- Javadoc - @Inject
- Baeldung - The @Resource Annotation
- Shell26 - @Autowired vs @Resource vs @Inject

Java-аннотация @Inject входит в пакет javax.inject и, чтобы её использовать, нужно добавить зависимость:

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

Размещается над полями, методами, и конструкторами с аргументами. @Inject как и @Autowired в первую очередь пытается подключить зависимость по типу, затем по описанию и только потом по имени. Это означает, что даже если имя переменной ссылки на класс отличается от имени компонента, но они одинакового типа, зависимость все равно будет разрешена:

```
@Inject
private ArbitraryDependency fieldInjectDependency;
```

отличается от имени компонента, настроенного в контексте приложения:

```
@Bean
public ArbitraryDependency injectDependency() {
    ArbitraryDependency injectDependency = new ArbitraryDependency();
    return injectDependency;
}
```

Разность имён injectDependency и fieldInjectDependency не имеет значения, зависимость будет подобрана по типу ArbitraryDependency.

Если в контейнере несколько бинов-кандидатов на внедрение, то нужно использовать аннотацию @Qualifier:

```
@Inject
@Qualifier("defaultFile")
private ArbitraryDependency defaultDependency;
@Inject
@Qualifier("namedFile")
private ArbitraryDependency namedDependency;
```

При использовании конкретного имени (Id) бина используем @Named:

```
@Inject
@Named("yetAnotherFieldInjectDependency")
private ArbitraryDependency yetAnotherFieldInjectDependency;
```

11. Расскажите про аннотацию @Lookup

Источники: Sysout - Как использовать аннотацию @Lookup

Обычно бины в приложении Spring являтся синглтонами, и для внедрения зависимостей мы используем конструктор или сеттер.

Но бывает и другая ситуация: имеется бин Car – синглтон (singleton bean), и ему требуется каждый раз новый экземпляр бина Passenger. То есть Car – синглтон, а Passenger – так называемый прототипный бин (prototype bean). Жизненные циклы бинов разные. Бин Car создается контейнером только раз, а бин Passenger создается каждый раз новый – допустим, это происходит каждый раз при вызове какого-то метода бина Car. Вот здесь-то и пригодится внедрение бина с помощью Lookup-метода. Оно происходит не при инициализации контейнера, а позднее: каждый раз, когда вызывается метод.

Суть в том, что мы создаём метод-заглушку в бине Car и помечаем его специальным образом – аннотацией @Lookup. Этот метод должен возвращать бин Passenger, каждый раз новый. Контейнер Spring под капотом создаст прокси-подкласс и переопределит этот метод и будет нам выдавать новый экземпляр бина Passenger при каждом вызове аннотированного метода. Даже если в нашей заглушке он возвращает null (а так и надо делать - всё равно этот метод будет переопределен в прокси-подклассе):

```
@Component
public class Car {
    @Lookup
    public Passenger createPassenger() {
        return null;
    }
    public String drive(String name) {
        Passenger passenger = createPassenger();
        passenger.setName(name);
        return "car with " + passenger.getName();
    }
}
```

Допустим, в бине есть метод drive(), и при каждом вызове метода drive() бину Car требуется новый экземпляр бина Passenger – сегодня пассажир Петя, завтра – Вася. То есть бин Passenger прототипный. Для получения этого бина надо написать метод-заглушку createPassenger() и аннотировать его с помощью @Lookup.

Контейнер Spring переопределит этот метод-заглушку и будет выдавать при его вызове каждый раз новый экземпляр Passenger.

Осталось только определить бин Passenger как прототипный:

```
@Component
@Scope("prototype")
public class Passenger {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
```

```
        this.name = name;
    }
}
```

Теперь при вызове метода drive() мы можем везти каждый раз нового пассажира. Имя его передаётся в аргументе метода drive(), и затем задается сеттером во вновь созданном экземпляре пассажира.

12. Можно ли вставить бин в статическое поле? Почему?

Источники:
- Mkyong - Inject value into static variables
- Stackoverflow - Can you use @Autowired with static fields?
- Stackoverflow - Why can't we autowire static fields in spring?

Spring не позволяет внедрять бины напрямую в статические поля, например:

```
@Component
public class TestDataInit {

    @Autowired
    private static OrderItemService orderItemService;
}
```

Если вы распечатаете TestDataInit.orderItemService, там будет null.

Чтобы исправить это, создайте нестатический сеттер-метод:

```
@Component
public class TestDataInit {

    private static OrderItemService orderItemService;

    @Autowired
    public void setOrderItemService(OrderItemService orderItemService) {
        TestDataInit.orderItemService = orderItemService;
    }
}
```

13. Расскажите про аннотации @Primary и @Qualifier

Источники: Baeldung - Spring @Primary Annotation
Baeldung - Spring @Qualifier Annotation

Мы используем @Primary, чтобы отдавать предпочтение бину, когда есть несколько бинов одного типа. Эта аннотация полезна, когда мы хотим указать, какой компонент определенного типа должен внедряться по умолчанию.

```
@Configuration
public class Config {

    @Bean
    public Employee JohnEmployee() {
        return new Employee("John");
    }

    @Bean
    @Primary
    public Employee TonyEmployee() {
        return new Employee("Tony");
    }
}
```

или с аннотацией @Component

```
@Component
public class DepartmentManager implements Manager {
    @Override
    public String getManagerName() {
        return "Department manager";
    }
}
@Component
@Primary
public class GeneralManager implements Manager {
    @Override
    public String getManagerName() {
        return "General manager";
    }
}
```

Теперь, где будут требоваться бины типа Employee и Manager будут созданы и внедрены TonyEmployee и GeneralManager.

Когда есть несколько бинов одного типа, подходящих для внедрения, аннотация @Qualifier позволяет указать в качестве аргумента имя конкретного бина, который следует внедрить.

Стоит отметить, что если присутствуют аннотации @Qualifier и @Primary, то аннотация @Qualifier будет иметь приоритет. По сути, @Primary определяет значение по умолчанию, в то время как @Qualifier более специфичен.

14. Как заинжектить примитив?

Источники: Baeldung - Spring @Value Annotation

@Value

Внедрить в поле примитив можно с помощью аннотации @Value на уровне параметров поля или конструктора/метода.

Нам понадобится файл свойств (*.properties), чтобы определить значения, которые мы хотим внедрить аннотацией @Value. Сначала в нашем классе конфигурации нам нужно указать аннотацию @PropertySource с именем файла свойств.

```
@Component
@PropertySource("classpath:values.properties")
public class CollectionProvider {
    ...
}
```

Содержимое файла values.properties:

```
value.from.file=Value got from the file
priority=high
listOfValues=A,B,C
```

Внедряем значение value.from.file, равное “Value got from the file”:

```
@Value("${value.from.file}")
private String valueFromFile;
```

Если из файла не подтянутся значения по тем или иным причинам, то можно указать значения, которые будут внедрены по умолчанию. В данном примере, если не будет доступен value.from.file, то внедрится значение “some default”:

```
@Value("${value.from.file:some default}")
private String someDefault;
```

Если нужно внедрить несколько значений, то можно их определить в файле *.properties через запятую и Spring внедрит их как массив:

```
@Value("${listOfValues}")
private String[] valuesArray;
```

@Value with SpEL

Кроме того, для внедрения значений мы можем использовать язык SpEL (Spring Expression Language):

```
@Value("#{systemProperties['priority']}")
private String spelValue;
```

или со значениями по умолчанию:

```
@Value("#{systemProperties['unknown'] ?: 'some default'}")
private String spelSomeDefault;
```

Мы можем использовать значение поля из другого бина. Предположим, у нас есть бин с именем someBean с полем someValue, равным 10. Тогда в этом примере в поле будет записано число 10:

```
@Value("#{someBean.someValue}")
private Integer someBeanValue;
```

Мы можем манипулировать свойствами, чтобы получить список значений. В следующем примере мы получаем список строковых значений A, B и C:

```
@Value("#{'${listOfValues}'.split(',')}")
private List<String> valuesList;
```

@Value with Map

Мы также можем использовать аннотацию @Value для добавления свойств в Map. Для начала нам нужно определить свойство в формате {key: ‘value '} в нашем файле свойств:

```
valuesMap={key1: '1', key2: '2', key3: '3'}
```

Теперь мы можем вставить это значение из файла свойств в виде карты:

```
@Value("#{${valuesMap}}")
private Map<String, Integer> valuesMap;
```

Можем просто внедрить значение по ключу:

```
@Value("#{${valuesMap}.key1}")
private Integer valuesMapKey1;
```

Если мы не уверены, содержит ли Map определенный ключ, мы должны выбрать более безопасное выражение, которое не будет генерировать исключение, а установит значение в null, если ключ не найден:

```
@Value("#{${valuesMap}['unknownKey']}")
private Integer unknownMapKey;
```

Мы также можем установить значения по умолчанию для свойств или ключей, которые могут не существовать:

```
@Value("#{${unknownMap : {key1: '1', key2: '2'}}}")
private Map<String, Integer> unknownMap;
@Value("#{${valuesMap}['unknownKey'] ?: 5}")
private Integer unknownMapKeyWithDefaultValue;
```

Записи карты также могут быть отфильтрованы перед внедрением. Предположим, нам нужно получить только те записи, значения которых больше единицы:

```
@Value("#{${valuesMap}.?[value>'1']}")
private Map<String, Integer> valuesMapFiltered;
```

Мы также можем использовать аннотацию @Value для добавления всех текущих системных свойств:

```
@Value("#{systemProperties}")
private Map<String, String> systemPropertiesMap;
```

@Value with Constructor

Мы можем внедрять значения в конструкторе, если оно не найдено, то будет внедрено значение по умолчанию:

```
@Component
@PropertySource("classpath:values.properties")
public class PriorityProvider {
```

```
    private String priority;

    @Autowired
    public PriorityProvider(@Value("${priority:normal}") String priority) {
        this.priority = priority;
    }

    // standard getter
}
```

@Value with Setter

В приведенном коде мы используем выражение SpEL для добавления списка значений в метод setValues:

```
@Component
@PropertySource("classpath:values.properties")
public class CollectionProvider {

    private List<String> values = new ArrayList<>();

    @Autowired
    public void setValues(
                @Value("#{'${listOfValues}'.split(',')}") List<String> values) {
        this.values.addAll(values);
    }
    // standard getter
}
```

15. Как заинжектить коллекцию?

Источники: LogicBig - Spring - Injecting Arrays and Collections
Habr - Некоторые тонкости injection'а коллекций в Spring'е

Array Injection

Мы можем вставлять массивы примитивов и ссылочных типов. Со всеми массивами и коллекциями мы можем использовать внедрение через конструкторы, сеттеры или поля.

```
@Configuration
public class ArrayExample {
    @Bean
    public TestBean testBean () {
        return new TestBean();
    }
    @Bean
    public String[] strArray(){
        return new String[]{"two", "three", "four"};
}

public class TestBean {
    private  String[] stringArray;

    @Autowired
    public void setStringArray (String[] stringArray) {
        this.stringArray = stringArray;
    }
}
```

Collections Injection

```
public class ListExample {
    @Bean
    public TestBean testBean () {
        return new TestBean();
    }
    @Bean
    public List<String> strList() {
        return Arrays.asList("two", "three", "four");
    }
}
public class TestBean {
    private  List<String> stringList;

    @Autowired
    public void setStringList (List<String> stringList) {
        this.stringList = stringList;
    }
}
```

Коллекции бинов одного типа

Также мы можем собрать все бины одного типа, находящиеся в контейнере, и внедрить их в коллекцию или массив:

```
@Configuration
public class SetInjection {
    @Bean
    public TestBean testBean () {
        return new TestBean();
    }
    @Bean
    public RefBean refBean1 () {
        return new RefBean("bean 1");
    }
    @Bean
    public RefBean refBean2 () {
        return new RefBean2("bean 2");
    }
    @Bean
    public RefBean refBean3 () {
        return new RefBean3("bean 3");
    }
    public static class TestBean {
        // All bean instances of type RefBean will be injecting here
        @Autowired
        private Set<RefBean> refBeans;
            ...
    }
    public static class RefBean{
        private String str;
            public RefBean(String str){
             this.str = str;
            }
         .....
    }
}
```

Если мы хотим внедрить вышеупомянутые бины RefBean в Map, то значениями Map будут сами бины, а ключами будут имена бинов:

```
{refBean1 = RefBean{str='bean 1'}, refBean2 = RefBean{str='bean 2'}, refBean3 =
RefBean{str='bean 3'}}
```

Использование @Qualifier

Методы класса JavaConfig (те, которые аннотированы @Bean) могут быть объявлены с определенным квалифицирующим типом, используя @Qualifier. Мы использовали параметр 'name' у аннотации @Bean, чтобы указать конкретный классификатор для бина. Но элемент 'name', на самом деле, является не столько именем, сколько идентификатором бина, который должен быть уникальным, потому что все бины хранятся в контейнере в Map.

В случае с коллекцией мы хотим, чтобы несколько бинов имели одно и то же имя квалификатора, чтобы их можно было внедрить в одну коллекцию с одним и тем же

квалификатором. В этом случае мы должны использовать аннотацию @Qualifier вместе с @Bean вместо элемента name.

```
@Configuration
public class SetInjection {
    @Bean
    public TestBean testBean () {
        return new TestBean();
    }
    @Bean
    public RefBean refBean1 () {
        return new RefBean("bean 1");
    }
    @Bean
    @Qualifier("myRefBean")
    public RefBean refBean2 () {
        return new RefBean2("bean 2");
    }
    @Bean
    @Qualifier("myRefBean")
    public RefBean refBean3 () {
        return new RefBean3("bean 3");
    }
    public static class TestBean {
        @Autowired
        @Qualifier("myRefBean")
        private Set<RefBean> refBeans;
    }
}
```

Только бины с именами refBean2 и refBean3 попадут в коллекцию, так как у них одинаковые квалификаторы - myRefBean.

Упорядочивание элементов массивов / списков

Бины могут быть упорядочены, когда они вставляются в списки (не Set или Map) или массивы. Поддерживаются как аннотация @Order, так и интерфейс Ordered. Например:

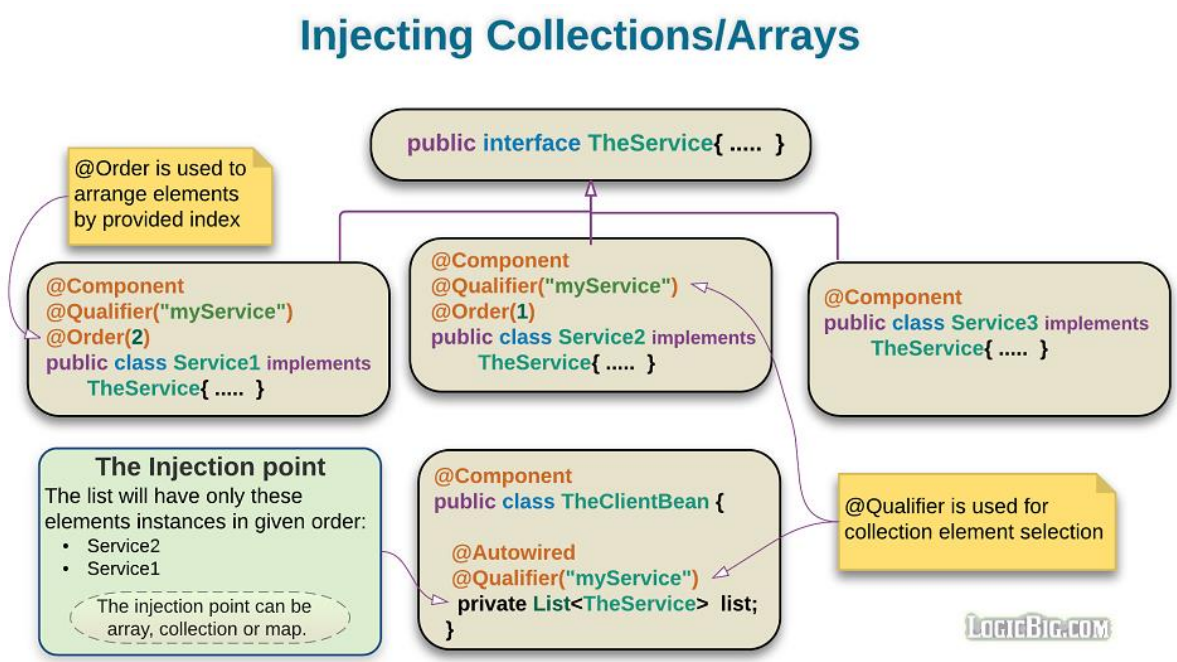
```
@Configuration
public class ArrayExample {
    @Bean
    public TestBean testBean () {
        return new TestBean();
    }
    @Bean
    @Order(3)
    public String refString1 () {
        return "my string 1";
    }
    @Bean
    @Order(1)
    public String refString2 () {
        return "my string 2";
    }
```

```
    @Bean
    @Order(2)
    public String refString3 () {
        return "my string 3";
    }
    private static class TestBean {
        private String[] stringArray;

        @Autowired
        public void setStringArray (String[] stringArray) {
            this.stringArray = stringArray;
        }
        public String[] getStringArray () {
            return stringArray;
        }
    }
}
```

Массив строк будет выглядеть так:

```
[my string 2, my string 3, my string 1]
```



Также мы можем объявить бин-коллекцию и внедрять её в другие бины:

```
@Service
@Getter
public class ActionHeroesService {
    @Autowired
    List<Hero> actionHeroes;
}

@Configuration
public class HeroesConfig {
```

```
    @Bean
    public List<Hero> action() {
        List<Hero> result = new ArrayList<>();
        result.add(new Terminator());
        result.add(new Rambo());
        return result;
    }
}
```

16. Расскажите про аннотацию @Conditional

Источники:
- Spring API - @Conditional
- Spring API - Interface Condition
- Spring - Conditionally Include @Configuration Classes or @Bean Methods
- Baeldung - Custom Auto-Configuration with Spring Boot
- Habr - Использование Conditional в Spring

Часто бывает полезно включить или отключить весь класс @Configuration, @Component или отдельные методы @Bean в зависимости от каких-либо условий.

Аннотация @Conditional указывает, что компонент имеет право на регистрацию в контексте только тогда, когда все условия соответствуют. Может применяться:

- над классами прямо или косвенно аннотированными @Component, включая классы @Configuration;
- над методами @Bean;
- как мета-аннотация при создании наших собственных аннотаций-условий.

Условия проверяются непосредственно перед тем, как должно быть зарегистрировано BeanDefinition компонента, и они могут помешать регистрации данного BeanDefinition. Поэтому нельзя допускать, чтобы при проверке условий мы взаимодействовали с бинами (которых еще не существует), с их BeanDefinition-ами можно.

Условия мы определяем в специально создаваемых нами классах, которые должны имплементировать функциональный интерфейс Condition с одним единственным методом, возвращающим true или false:

```
boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata)
```

Создав свой класс и переопределив в нем метод matches() с нашей логикой, мы должны передать этот класс в аннотацию @Conditional в качестве параметра:

```
@Configuration
@Conditional(OurConditionClass.class)
class MySQLAutoconfiguration {
    //...
}
```

Для того, чтобы проверить несколько условий, можно передать в @Conditional несколько классов с условиями:

```
@Bean
@Conditional(HibernateCondition.class, OurConditionClass.class)
Properties additionalProperties() {
    //...
}
```

Если класс @Configuration помечен как @Conditional, то на все методы @Bean, аннотации @Import и аннотации @ComponentScan, связанные с этим классом, также будут распространяться указанные условия.

Для более детальной настройки классов, аннотированных @Configuration, предлагается использовать интерфейс ConfigurationCondition.

В одном классе - одно условие. Для создания более сложных условий можно использовать классы AnyNestedCondition, AllNestedConditions и NoneNestedConditions.

В Spring Framework имеется множество готовых аннотаций (и связанных с ними склассами-условиями, имплементирующими интерфейс Condition), которые можно применять совместно над одним определением бина:

Аннотация	**Описание**
ConditionalOnBean	Условие выполняется, в случае если присутствует нужный бин в BeanFactory.
ConditionalOnClass	Условие выполняется, если нужный класс есть в classpath.
ConditionalOnCloudPlatform	Условие выполняется, когда активна определенная платформа.
ConditionalOnExpression	Условие выполняется, когда SpEL выражение вернуло положительное значение.
ConditionalOnJava	Условие выполняется, когда приложение запущено с определенной версией JVM.
ConditionalOnJndi	Условие выполняется, только если через JNDI доступен определенный ресурс.
ConditionalOnMissingBean	Условие выполняется, в случае если нужный бин отсутствует в контейнере.
ConditionalOnMissingClass	Условие выполняется, если нужный класс отсутствует в classpath.
ConditionalOnNotWebApplication	Условие выполняется, если контекст приложения не является веб контекстом.
ConditionalOnProperty	Условие выполняется, если в файле настроек заданы нужные параметры.
ConditionalOnResource	Условие выполняется, если присутствует нужный ресурс в classpath.
ConditionalOnSingleCandidate	Условие выполняется, если bean-компонент указанного класса уже содержится в контейнере и он единственный.
ConditionalOnWebApplication	Условие выполняется, если контекст приложения является веб контекстом.

17. Расскажите про аннотацию @ComponentScan

Источники: Spring - Automatically Detecting Classes and Registering Bean Definitions
Baeldung - Spring Component Scanning
Baeldung - Spring @ComponentScan – Filter Types

Аннотация @ComponentScan используется вместе с аннотацией @Configuration для указания пакетов, которые мы хотим сканировать на наличие компонентов, из которых нужно сделать бины.

@ComponentScan без аргументов указывает Spring по умолчанию сканировать текущий пакет и все его подпакеты. Текущий пакет - тот, в котором находится файл конфигурации с этой самой аннотацией @ComponentScan. В данном случае в контейнер попадут:

- бин конфигурационного класса;
- бины, объявленные в конфигурационном классе с помощью @Bean;
- все бины из пакета и его подпакетов.

Аннотация @SpringBootApplication включает в себя аннотации @ComponentScan, @SpringBootConfiguration и @EnableAutoConfiguration, но это не мешает разместить её ещё раз отдельно для указания конкретного пакета.

Если указать @ComponentScan с атрибутом basePackages, то это изменит пакет по умолчанию на указанный:

```
@ComponentScan(basePackages =
               "com.baeldung.componentscan.springapp.animals")
@Configuration
public class SpringComponentScanApp {
   // ...
}
```

Если указать @ComponentScan с атрибутом excludeFilters, то это позволит использовать фильтр и исключить ненужные классы из процесса сканирования:

```
@ComponentScan(excludeFilters =
    @ComponentScan.Filter(type=FilterType.REGEX,
        pattern="com\\.baeldung\\.componentscan\\.springapp\\.flowers\\..*"))
```

18. Расскажите про аннотацию @Profile

Источники:
Spring - Bean Definition Profiles
Spring API - @Profile
Baeldung - Spring Profiles
Mkyong - Spring Profiles example

Профили — это ключевая особенность Spring Framework, позволяющая нам относить наши бины к разным профилям (логическим группам), например, dev, test, prod.

Мы можем активировать разные профили в разных средах, чтобы загрузить только те бины, которые нам нужны.

Используя аннотацию @Profile, мы относим бин к конкретному профилю. Её можно применять на уровне класса или метода. Аннотация @Profile принимает в качестве аргумента имя одного или нескольких профилей. Она фактически реализована с помощью гораздо более гибкой аннотации @Conditional.

Рассмотрим базовый сценарий - у нас есть бин, который должен быть активным только во время разработки, но не должен использоваться в продакшене. Мы аннотируем этот компонент с профилем «dev», и он будет присутствовать в контейнере только во время разработки - во время продакшена профиль dev просто не будет активен:

```
@Component
@Profile("dev")
public class DevDatasourceConfig
```

В качестве быстрого обозначения имена профилей также могут начинаться с оператора NOT, например «!dev», чтобы исключить их из профиля. В приведенном ниже примере компонент активируется, только если профиль «dev» не активен:

```
@Component
@Profile("!dev")
public class DevDatasourceConfig
```

Следующим шагом является активация нужного профиля для того, чтобы в контейнере были зарегистрированы только бины, соответствующие данному профилю. Одновременно могут быть активны несколько профилей.

По умолчанию, если профиль бина не определен, то он относится к профилю “default”. Spring также предоставляет способ установить профиль по умолчанию, когда другой профиль не активен, используя свойство «spring.profiles.default».

В Spring Boot есть возможность иметь один файл настроек application.properties, в котором будут основные настройки для всех профилей, и иметь по файлу настроек для каждого профиля application-dev.properties и application-prod.properties, содержащие свои собственные дополнительные настройки.

19. Расскажите про ApplicationContext и BeanFactory, чем отличаются? В каких случаях что стоит использовать?

Источники:
Spring - The IoC Container
Spring - BeanFactory or ApplicationContext?
Baeldung - Difference Between BeanFactory and ApplicationContext

BeanFactory

BeanFactory — это интерфейс, который предоставляет механизм конфигурации, способный управлять объектами любого типа. В общем, BeanFactory предоставляет инфраструктуру конфигурации и основные функциональные возможности.

BeanFactory легче по сравнению с ApplicationContext.

ApplicationContext

ApplicationContext является наследником BeanFactory и полностью реализует его функционал, добавляя больше специфических enterprise-функций.

ApplicationContext vs. BeanFactory

Feature	BeanFactory	ApplicationContext
Bean instantiation/wiring	Yes	Yes
Integrated lifecycle management	No	Yes
Automatic BeanPostProcessor registration	No	Yes
Automatic BeanFactoryPostProcessor registration	No	Yes
Convenient MessageSource access (for internalization)	No	Yes
Built-in ApplicationEvent publication mechanism	No	Yes

1. ApplicationContext загружает все бины при запуске, а BeanFactory - по требованию.
2. ApplicationContext расширяет BeanFactory и предоставляет функции, которые подходят для корпоративных приложений:
    a. поддержка внедрения зависимостей на основе аннотаций;
    b. удобный доступ к MessageSource (для использования в интернационализации);
    c. публикация ApplicationEvent - для бинов, реализующих интерфейс ApplicationListener, с помощью интерфейса ApplicationEventPublisher;
    d. простая интеграция с функциями Spring AOP.
3. ApplicationContext поддерживает автоматическую регистрацию BeanPostProcessor и BeanFactoryPostProcessor. Поэтому всегда желательно использовать ApplicationContext, потому что Spring 2.0 (и выше) интенсивно использует BeanPostProcessor.

4. ApplicationContext поддерживает практически все типы scope для бинов, а BeanFactory поддерживает только два - Singleton и Prototype.
5. В BeanFactory не будут работать транзакции и Spring AOP. Это может привести к путанице, потому что конфигурация с виду будет корректной.

20. Расскажите про жизненный цикл бина, аннотации @PostConstruct и @PreDestroy()

Источники:
Spring - Introduction to the Spring IoC Container and Beans
Spring API - BeanPostProcessor
Shell26 - Bean
Baeldung - PostConstruct and PreDestroy Annotations
Spring-projects.ru - Урок 2: Введение в Spring IoC контейнер
Habr - Этапы инициализации контекста
Medium - Spring под капотом
YouTube - Евгений Борисов — Spring-потрошитель, часть 1
YouTube - Евгений Борисов — Spring-потрошитель, часть 2

Жизненный цикл бинов

1. Парсирование конфигурации и создание BeanDefinition

Цель первого этапа — это создание всех BeanDefinition. Объекты BeanDefinition — это набор метаданных будущего бина, макет, по которому нужно будет создавать бин в случае необходимости. То есть для каждого бина создается свой объект BeanDefinition, в котором хранится описание того, как создавать и управлять этим конкретным бином. Проще говоря, сколько бинов в программе - столько и объектов BeanDefinition, их описывающих.

BeanDefinition содержат (среди прочего) следующие метаданные:

1. Имя класса с указанием пакета: обычно это фактический класс бина.
2. Элементы поведенческой конфигурации бина, которые определяют, как бин должен вести себя в контейнере (scope, обратные вызовы жизненного цикла и т.д.).
3. Ссылки на другие bean-компоненты, которые необходимы для его работы. Эти ссылки также называются зависимостями.
4. Другие параметры конфигурации для установки во вновь созданном объекте - например, ограничение размера пула или количество соединений, используемых в бине, который управляет пулом соединений.

Эти метаданные преобразуются в набор свойств, которые составляют каждое BeanDefinition. В следующей таблице описаны эти свойства:

Свойство	**Ссылка с описанием**
Class	Instantiating Beans
Name	Naming Beans
Scope	Bean Scopes
Constructor arguments	Dependency Injection
Properties	Dependency Injection
Autowiring mode	Autowiring Collaborators

Lazy initialization mode	Lazy-initialized Beans
Initialization method	Initialization Callbacks
Destruction method	Destruction Callbacks

При конфигурации через аннотации с указанием пакета для сканирования или JavaConfig используется класс AnnotationConfigApplicationContext. Регистрируются все классы с @Configuration для дальнейшего парсирования, затем регистрируется специальный BeanFactoryPostProcessor, а именно BeanDefinitionRegistryPostProcessor, который при помощи класса ConfigurationClassParser парсирует JavaConfig, загружает описания бинов (BeanDefinition), создаёт граф зависимостей (между бинами) и создаёт:

```
Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<>(256);
```

в которой хранятся все описания бинов, обнаруженных в ходе парсинга конфигурации.

2. Настройка созданных BeanDefinition

После первого этапа у нас имеется коллекция Map, в которой хранятся BeanDefinition-ы. BeanFactoryPostProcessor-ы на этапе создания BeanDefinition-ов могут их настроить как нам необходимо. BeanFactoryPostProcessor-ы могут даже настроить саму BeanFactory ещё до того, как она начнет работу по созданию бинов. В интерфейсе BeanFactoryPostProcessor всего один метод:

```
public interface BeanFactoryPostProcessor {
      void postProcessBeanFactory(ConfigurableListableBeanFactory
                                  beanFactory) throws BeansException;
}
```

3. Создание кастомных FactoryBean (только для XML-конфигурации)

4. Создание экземпляров бинов

Сначала BeanFactory из коллекции Map с объектами BeanDefinition достаёт те из них, из которых создаёт все BeanPostProcessor-ы, необходимые для настройки обычных бинов.

Создаются экземпляры бинов через BeanFactory на основе ранее созданных BeanDefinition.

5. Настройка созданных бинов

На данном этапе бины уже созданы, мы можем лишь их донастроить.

Интерфейс BeanPostProcessor позволяет вклиниться в процесс настройки наших бинов до того, как они попадут в контейнер. ApplicationContext автоматически обнаруживает любые бины с реализацией BeanPostProcessor и помечает их как “post-processors” для того, чтобы создать их определенным способом. Например, в Spring есть реализации BeanPostProcessor-ов, которые обрабатывают аннотации @Autowired, @Inject, @Value и @Resource.

Интерфейс несет в себе два метода: postProcessBeforeInitialization(Object bean, String beanName) и postProcessAfterInitialization(Object bean, String beanName). У обоих методов параметры абсолютно одинаковые. Разница только в порядке их вызова. Первый вызывается до init-метода, второй - после.

Как правило, BeanPostProcessor-ы, которые заполняют бины через маркерные интерфейсы или тому подобное, реализовывают метод postProcessBeforeInitialization (Object bean, String beanName), тогда как BeanPostProcessor-ы, которые оборачивают бины в прокси, обычно реализуют postProcessAfterInitialization (Object bean, String beanName).

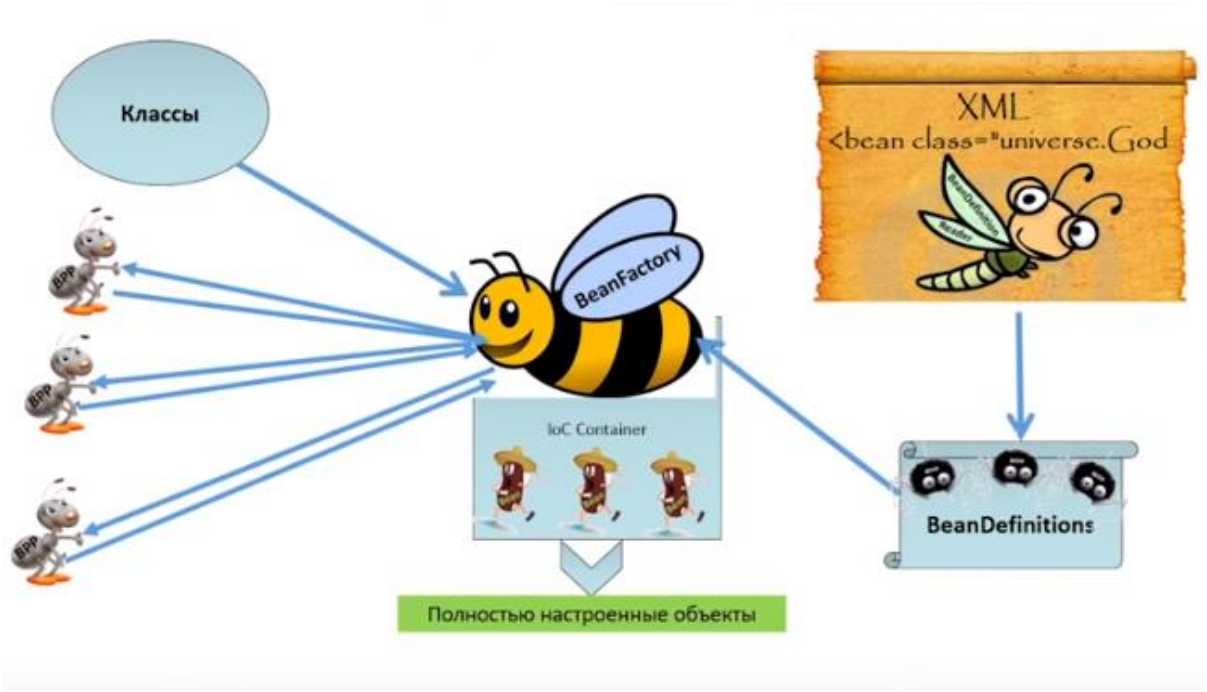
Прокси — это класс-декорация над бином. Например, мы хотим добавить логику нашему бину, но джава-код уже скомпилирован, поэтому нам нужно на лету сгенерировать новый класс. Этим классом мы должны заменить оригинальный класс так, чтобы никто не заметил подмены. Есть два варианта создания этого класса:

1. либо он должен наследоваться от оригинального класса (CGLIB) и переопределять его методы, добавляя нужную логику;
2. либо он должен имплементировать те же самые интерфейсы, что и первый класс (Dynamic Proxy).

По конвенции спринга, если какой-то из BeanPostProcessor-ов меняет что-то в классе, то он должен это делать на этапе postProcessAfterInitialization(). Таким образом мы уверены, что initMethod у данного бина, работает на оригинальный метод, до того, как на него накрутился прокси.

Хронология событий:

1. Сначала сработает метод postProcessBeforeInitialization() всех имеющихся BeanPostProcessor-ов.
2. Затем, при наличии, будет вызван метод, аннотированный @PostConstruct.
3. Если бин имплементирует InitializingBean, то Spring вызовет метод afterPropertiesSet() - *не рекомендуется к использованию как устаревший.*
4. При наличии, будет вызван метод, указанный в параметре initMethod аннотации @Bean.
5. В конце бины пройдут через postProcessAfterInitialization (Object bean, String beanName). **Именно на данном этапе создаются прокси стандартными BeanPostProcessor-ами.** Затем отработают наши кастомные BeanPostProcessor-ы и применят нашу логику к прокси-объектам. После чего все бины окажутся в контейнере, который будет обязательно обновлен методом refresh().
6. Но даже после этого мы можем донастроить наши бины ApplicationListener-ами.
7. Теперь всё.



6. Бины готовы к использованию

Их можно получить с помощью метода ApplicationContext#getBean().

7. Закрытие контекста

Когда контекст закрывается (метод close() из ApplicationContext), бин уничтожается.

Если в бине есть метод, аннотированный @PreDestroy, то перед уничтожением вызовется этот метод.

Если бин имплементирует DisposibleBean, то Spring вызовет метод destroy() - *не рекомендуется к использованию как устаревший*.

Если в аннотации @Bean определен метод destroyMethod, то будет вызван и он.

@PostConstruct

Spring вызывает методы, аннотированные @PostConstruct, только один раз, сразу после инициализации свойств компонента. За данную аннотацию отвечает один из BeanPostProcessor-ов.

Метод, аннотированный @PostConstruct, может иметь любой уровень доступа, может иметь любой тип возвращаемого значения (хотя тип возвращаемого значения игнорируется Spring-ом), метод не должен принимать аргументы. Он также может быть статическим, но преимуществ такого использования метода нет, т.к. доступ у него будет только к статическим полям/методам бина, и в таком случае смысл его использования для настройки бина пропадает.

Одним из примеров использования @PostConstruct является заполнение базы данных. Например, во время разработки нам может потребоваться создать пользователей по умолчанию.

@PreDestroy

Метод, аннотированный @PreDestroy, запускается только один раз, непосредственно перед тем, как Spring удаляет наш компонент из контекста приложения.

Как и в случае с @PostConstruct, методы, аннотированные @PreDestroy, могут иметь любой уровень доступа, но не могут быть статическими.

Целью этого метода может быть освобождение ресурсов или выполнение любых других задач очистки до уничтожения бина, например, закрытие соединения с базой данных.

Обратите внимание, что аннотации @PostConstruct и @PreDestroy являются частью Java EE, а именно пакета javax.annotation модуля java.xml.ws.annotation. И поскольку Java EE устарела в Java 9, то с этой версии пакет считается устаревшим (Deprecated). С Java 11 данный пакет вообще удален, поэтому мы должны добавить дополнительную зависимость для использования этих аннотаций:

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

21. Расскажите про scope бинов. Какой scope используется по умолчанию? Что изменилось в пятом Spring?

Источники: Spring - Bean Scopes
Mkyong - Spring 5 – Bean scopes
YouTube - Alishev Урок 8: Жизненный цикл бина

Spring Framework поддерживает шесть scopes:

1. singleton
2. prototype
3. request
4. session
5. application
6. websocket
7. не активированный по умолчанию Custom thread scope.

С 3 по 6 доступны только в веб-приложениях. Мы также можем создать свой собственный scope.

Singleton

Является дефолтным scope. В контейнере будет создан только один бин, и все запросы на него будут возвращать один и тот же бин. Этот бин хранится в контейнере, и все запросы и ссылки на этот бин возвращают закэшированный экземпляр.

Prototype

Scope “prototype” приводит к созданию нового бина каждый раз, когда он запрашивается.

Для бинов со scope “prototype” Spring не вызывает метод destroy(). Spring не берет на себя контроль полного жизненного цикла бина со scope @prototype”. Spring не хранит такие бины в своём контексте (контейнере), а отдаёт их клиенту и больше о них не заботится (в отличие от синглтон-бинов).

Request

Контейнер создает новый экземпляр для каждого HTTP-запроса. Таким образом, если сервер в настоящее время обрабатывает 50 запросов, тогда контейнер может иметь не более 50 бинов, по одному для каждого HTTP-запроса. Любое изменение состояния одного экземпляра не будет видимо другим экземплярам. Эти экземпляры уничтожаются, как только HTTP-запрос завершен.

```
@Component
@Scope("request")
public class BeanClass {
}

//or

@Component
```

```
@RequestScope
public class BeanClass {
}
```

Session

Бин создается в одном экземпляре для одной HTTP-сессии. Таким образом, если сервер имеет 20 активных сессий, тогда контейнер может иметь не более 20 бинов, по одному для каждой сессии. Все HTTP-запросы в пределах времени жизни одной сессии будут иметь доступ к одному и тому же бину.

```
@Component
@Scope("session")
public class BeanClass {
}

//or

@Component
@SessionScope
public class BeanClass {
}
```

Application

Бин со scope “application” создается в одном экземпляре для жизненного цикла ServletContext. Виден как атрибут ServletContext. Синглтон - в одном экземпляре для ApplicationContext.

```
@Component
@Scope("application")
public class BeanClass {
}

//or

@Component
@ApplicationScope
public class BeanClass {
}
```

Websocket

Бин со scope “websocket” создается в одном экземпляре для определенного сеанса WebSocket. Один и тот же бин возвращается всякий раз, когда к нему обращаются в течение всего сеанса WebSocket.

```
@Component
@Scope("websocket")
public class BeanClass {
}
```

Custom thread scope

Spring по умолчанию не предоставляет thread scope, но его можно активировать. Каждый запрос на бин в рамках одного потока будет возвращать один и тот же бин.

В пятой версии Spring Framework не стало Global session scope.

22. Что такое АОП? Как реализовано в спринге?

Источники: Habr - Аспектно-ориентированное программирование, Spring AOP
Введение в AOP в Spring Boot
Shell26

Аспектно-ориентированное программирование (АОП) — это парадигма программирования, целью которой является повышение модульности за счет разделения междисциплинарных задач. Это достигается путем добавления дополнительного поведения к существующему коду без изменения самого кода.

АОП предоставляет возможность реализации в одном месте сквозной логики - т.е. логики, которая применяется к множеству частей приложения - и обеспечения автоматического применения этой логики по всему приложению.

Подход Spring к АОП заключается в создании "динамических прокси" для целевых объектов и "привязывании" объектов к конфигурированному совету для выполнения сквозной логики.

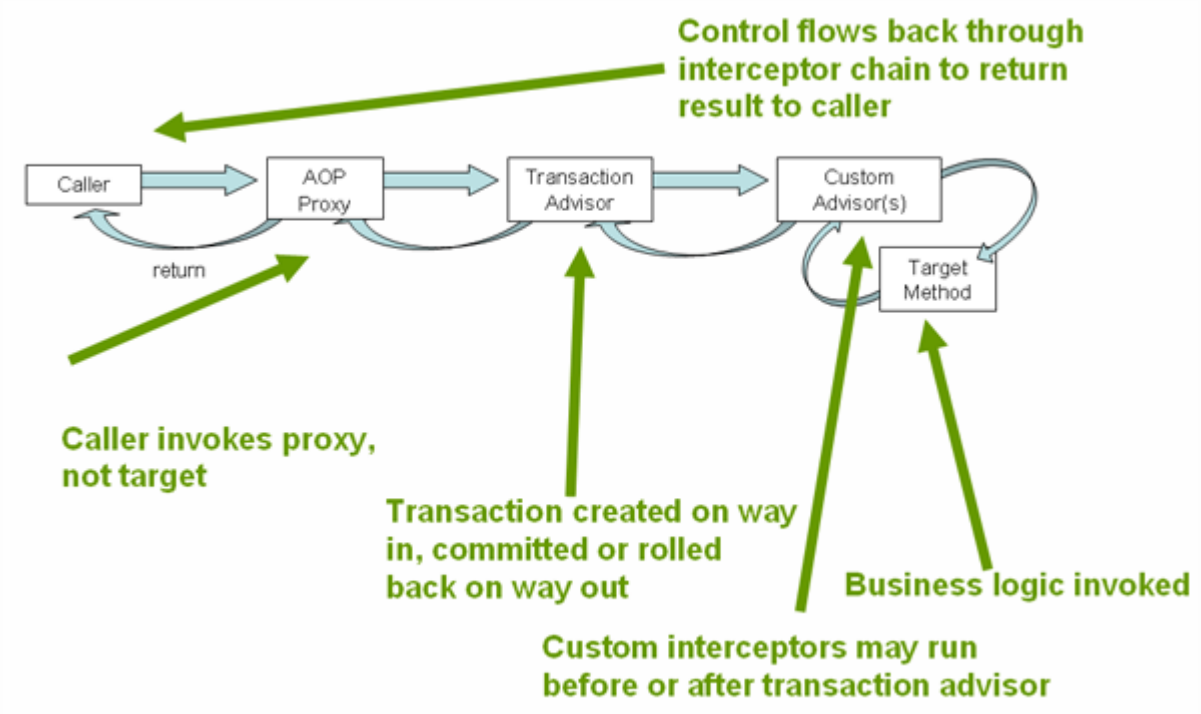
Прочитать статью.

23. Как спринг работает с транзакциями? Расскажите про аннотацию @Transactional

Источники:

Spring - Declarative transaction management
Spring - Using @Transactional
Spring API - @EnableTransactionManagement
Spring API - @Transactional
YouTube - Spring Framework - работаем с транзакциями
Baeldung - Transactions with Spring and JPA
Habr - Эффективное управление транзакциями в Spring
Akorsa - Как на самом деле работает @Transactional Spring?
Medium - Транзакции в Spring Framework
Shell26 - Как управлять транзакциями в Spring

Для работы с транзакциями Spring Framework использует AOP-прокси:



Для включения возможности управления транзакциями первым делом нужно разместить аннотацию @EnableTransactionManagement у класса-конфигурации @Configuration.

Аннотация @EnableTransactionManagement означает, что классы, помеченные @Transactional, должны быть обернуты аспектом транзакций. Однако, если мы используем Spring Boot и имеем зависимости spring-data-* или spring-tx, то управление транзакциями будет включено по умолчанию.

@EnableTransactionManagement отвечает за регистрацию необходимых компонентов Spring, таких как TransactionInterceptor и советы прокси (proxy advices- набор инструкций, выполняемых на точках среза - Pointcut). Регистрируемые компоненты помещают перехватчик в стек вызовов при вызове методов @Transactional.

Spring создает прокси для всех классов, помеченных @Transactional (либо если любой из методов класса помечен этой аннотацией). Прокси-объекты позволяют Spring Framework

вводить транзакционную логику до и после вызываемого метода - главным образом для запуска и коммита/отката транзакции.

Если мы разместим аннотацию @Transactional над классом @Service, то все его методы станут транзакционными. Так, при вызове, например, метода save() произойдет **примерно** следующее:

1. Вначале мы имеем:
   - класс TransactionInterceptor, у которого основной метод invoke(...), внутри которого вызывается метод класса-родителя TransactionAspectSupport: invokeWithinTransaction(...), в рамках которого происходит магия транзакций.
   - TransactionManager: решает, создавать ли новый EntityManager и/или транзакцию.
   - EntityManager proxy: EntityManager - это интерфейс, и то, что внедряется в бин в слое DAO на самом деле не является реализацией EntityManager. В это поле внедряется EntityManager proxy, который будет перехватывать обращение к полю EntityManager и делегировать выполнение конкретному EntityManager в рантайме. Обычно EntityManager proxy представлен классом SharedEntityManagerInvocationHandler.

2. Transaction Interceptor

   В TransactionInterceptor отработает код до работы метода save(), в котором будет определено, выполнить ли метод save() в пределах уже существующей транзакции БД или должна стартовать новая отдельная транзакция. TransactionInterceptor сам не содержит логики по принятию решения, решение начать новую транзакцию, если это нужно, делегируется TransactionManager. **Грубо говоря**, на данном этапе наш метод будет обёрнут в try-catch и будет добавлена логика до его вызова и после:

```
try {
   transaction.begin();
   // логика до
   service.save();
   // логика после
   transaction.commit();
} catch(Exception ex) {
  transaction.rollback();
  throw ex;
}
```

3. TransactionManager

   Менеджер транзакций должен предоставить ответ на два вопроса:
   - Должен ли создаться новый EntityManager?
   - Должна ли стартовать новая транзакция БД?

   TransactionManager принимает решение, основываясь на следующих фактах:
   - выполняется ли хоть одна транзакция в текущий момент или нет;
   - атрибута «propagation» у метода, аннотированного @Transactional (для примера, значение REQUIRES_NEW всегда стартует новую транзакцию).

   Если TransactionManager решил создать новую транзакцию, тогда:
   - Создается новый EntityManager;

- EntityManager «привязывается» к текущему потоку (Thread);
- «Получается» соединение из пула соединений БД;
- Соединение «привязывается» к текущему потоку.

И EntityManager и это соединение привязываются к текущему потоку, используя переменные ThreadLocal.

4. EntityManager proxy
   Когда метод save() слоя Service делает вызов метода save() слоя DAO, внутри которого вызывается, например, entityManager.persist(), то не происходит вызов метода persist() напрямую у EntityManager, записанного в поле класса DAO. Вместо этого метод вызывает EntityManager proxy, который достает текущий EntityManager для нашего потока, и у него вызывается метод persist().

5. Отрабатывает DAO-метод save().

6. TransactionInterceptor
   Отработает код после работы метода save(), а именно будет принято решение по коммиту/откату транзакции.

Кроме того, если мы в рамках одного метода сервиса обращаемся не только к методу save(), а к разным методам Service и DAO, то все они буду работать в рамках одной транзакции, которая оборачивает этот метод сервиса.

Вся работа происходит через прокси-объекты разных классов. Представим, что у нас в классе сервиса только один метод с аннотацией @Transactional, а остальные нет. Если мы вызовем метод с @Transactional, из которого вызовем метод без @Transactional, то оба будут отработаны в рамках прокси и будут обернуты в нашу транзакционную логику. Однако, если мы вызовем метод без @Transactional, из которого вызовем метод с @Transactional, то они уже не будут работать в рамках прокси и не будут обернуты в нашу транзакционную логику.

У @Transactional есть ряд параметров:

- @Transactional (isolation=Isolation.READ_COMMITTED) - уровень изоляции.
- @Transactional(timeout=60) - По умолчанию используется таймаут, установленный по умолчанию для базовой транзакционной системы. Сообщает TransactionManager-у о продолжительности времени, чтобы дождаться простоя транзакции, прежде чем принять решение об откате не отвечающих транзакций.
- @Transactional(propagation=Propagation.REQUIRED) - (Если не указано, распространяющееся поведение по умолчанию — REQUIRED.) Указывает, что целевой метод не может работать без другой транзакции. Если до вызова этого метода уже была запущена транзакция, то метод будет работать в той же транзакции, если транзакции не было, то будет создана новая.

  REQUIRES_NEW -Указывает, что новая транзакция должна запускаться каждый раз при вызове целевого метода. Если транзакция уже идет, она будет приостановлена, прежде чем будет запущена новая.

MANDATORY - Указывает, что для целевого метода требуется активная транзакция. Если активной транзакции нет, метод не сработает и будет выброшено исключение.
SUPPORTS - Указывает, что целевой метод может выполняться независимо от наличия транзакции. Если транзакция работает, он будет участвовать в той же транзакции. Если транзакции нет, он всё равно будет выполняться, если не будет ошибок. Методы, которые извлекают данные, являются лучшими кандидатами для этой опции.
NOT_SUPPORTED - Указывает, что целевой метод не требует распространения контекста транзакции. В основном те методы, которые выполняются в транзакции, но выполняют операции с оперативной памятью, являются лучшими кандидатами для этой опции.
NEVER - Указывает, что целевой метод вызовет исключение, если выполняется в транзакционном процессе. Этот вариант в большинстве случаев не используется в проектах.

- @Transactional (rollbackFor=Exception.class) - Значение по умолчанию: rollbackFor=RunTimeException.class В Spring все классы API бросают RuntimeException, это означает, что если какой-либо метод не выполняется, контейнер всегда откатывает текущую транзакцию. Проблема заключается только в проверяемых исключениях. Таким образом, этот параметр можно использовать для декларативного отката транзакции, если происходит Checked Exception.
- @Transactional (noRollbackFor=IllegalStateException.class) - Указывает, что откат не должен происходить, если целевой метод вызывает это исключение. Если внутри метода с @Transactional есть другой метод с аннотацией @Transactional (вложенная транзакция), то отработает только первая (в которую вложена), из-за особенностей создания proxy.

24. Расскажите про паттерн MVC, как он реализован в Spring?

Источники:
- Javastudy - Spring MVC – основные понятия, архитектура
- Medium - Introduction to Spring MVC (Part 1)
- Wiki - Model–view–controller
- Habr - Spring MVC — основные принципы
- Wiki - Spring MVC
- YouTube - Алишев Spring MVC. Конфигурация с помощью Java кода.
- Baeldung - Quick Guide to Spring Controllers
- Baeldung - Spring MVC Interview Questions
- Marcobehler - What is Spring MVC

MVC (Model-View-Controller)

Это шаблон проектирования программного обеспечения, который делит программную логику на три отдельных, но взаимосвязанных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо.

Модель (Model) предоставляет данные и реагирует на команды контроллера, изменяя своё состояние. Она содержит всю бизнес-логику приложения.

Представление (View) отвечает за отображение пользователю данных из модели в нужном формате.

Контроллер (Controller) содержит код, который отвечает за обработку действий пользователя и обменивается данными с моделью (любое действие пользователя в системе обрабатывается в контроллере).

Основная цель следования принципам MVC — отделить реализацию бизнес-логики приложения (модели) от ее визуализации (вида). Такое разделение повысит возможность повторного использования кода.

Польза применения MVC наиболее наглядна в случаях, когда пользователю нужно предоставлять одни и те же данные в разных формах. Например, в виде таблицы, графика или диаграммы (используя различные виды). При этом, не затрагивая реализацию видов, можно изменить реакции на действия пользователя (нажатие мышью на кнопке, ввод данных).

Spring Web MVC

Spring MVC - это оригинальный веб-фреймворк, основанный на Servlet API, предназначенный для создания веб-приложений на языке Java, с использованием двух самых популярных шаблонов проектирования - Front controller и MVC.

Front controller (Единая точка входа) - паттерн, где центральный сервлет, DispatcherServlet, принимает все запросы и распределяет их между контроллерами, обрабатывающими разные URL.

Spring MVC реализует четкое разделение задач, что позволяет нам легко разрабатывать и тестировать наши приложения. Данные задачи разбиты между разными компонентами: Dispatcher Servlet, Controllers, View Resolvers, Views, Models, ModelAndView, Model and Session Attributes, которые полностью независимы друг от друга, и отвечают только за одно направление. Поэтому MVC дает нам довольно большую гибкость. Он основан на интерфейсах (с предоставленными классами реализации), и мы можем настраивать каждую часть фреймворка с помощью пользовательских интерфейсов.

Основные интерфейсы для обработки запросов:

HandlerMapping. По запросу определяет, какие перехватчики (interceptors) с пре- и пост-процессорной обработкой запроса должны отработать, а затем решает, какому контроллеру (обработчику) нужно передать данный запрос на исполнение. Процесс их определения основан на некоторых критериях, детали которых зависят от реализации HandlerMapping.

Двумя основными реализациями HandlerMapping являются RequestMappingHandlerMapping (который поддерживает аннотированные методы @RequestMapping) и SimpleUrlHandlerMapping (который поддерживает явную регистрацию путей URI для обработчиков).

HandlerAdapter. Помогает DispatcherServlet вызвать обработчик, сопоставленный с запросом. Для вызова аннотированного контроллера необходимо прочитать аннотации над методами контроллера и принять решение. Основная цель HandlerAdapter - избавить DispatcherServlet от этой рутины.

ViewResolver. Сопоставляет имена представлений, возвращаемых методами контроллеров, с фактическими представлениями (html-файлами).

View. Отвечает за возвращение ответа клиенту в виде текстов и изображений. Используются встраиваемые шаблонизаторы (Thymeleaf, FreeMarker и т.д.), так как у Spring нет родных. Некоторые запросы могут идти прямо во View, не заходя в Model, другие проходят через все слои.

LocaleResolver. Определение часового пояса и языка клиента для того, чтобы предложить представления на его языке.

MultipartResolver. Обеспечивает Upload — загрузку на сервер локальных файлов клиента. По умолчанию этот интерфейс не включается в приложении и необходимо указывать его в файле конфигурации. После настройки любой запрос о загрузке будет отправляться этому интерфейсу.

FlashMapManager. Сохраняет и извлекает «входной» и «выходной» FlashMap, который можно использовать для передачи атрибутов из одного запроса в другой, обычно через редирект.

Ниже приведена последовательность событий, соответствующая входящему HTTP-запросу:

- После получения HTTP-запроса DispatcherServlet обращается к интерфейсу **HandlerMapping**, который определяет, какой **Контроллер (Controller)** должен быть вызван, после чего **HandlerAdapter**, отправляет запрос в нужный метод Контроллера.
- Контроллер принимает запрос и вызывает соответствующий служебный метод, основанный на GET, POST и т.д. Вызванный метод формирует данные **Модели** (например, набор данных из БД) и возвращает их в DispatcherServlet вместе с именем **Представления (View)** (как правило имя html-файла).
- При помощи интерфейса **ViewResolver** DispatcherServlet определяет, какое Представление нужно использовать на основании полученного имени и получает в ответе имя представления View.
  - если это REST-запрос на сырые данные (JSON/XML), то DispatcherServlet сам его отправляет;
  - если обычный запрос, то DispatcherServlet отправляет данные Модели в виде атрибутов в Представление (View) - шаблонизаторы Thymeleaf, FreeMarker и т.д., которые сами отправляют ответ.

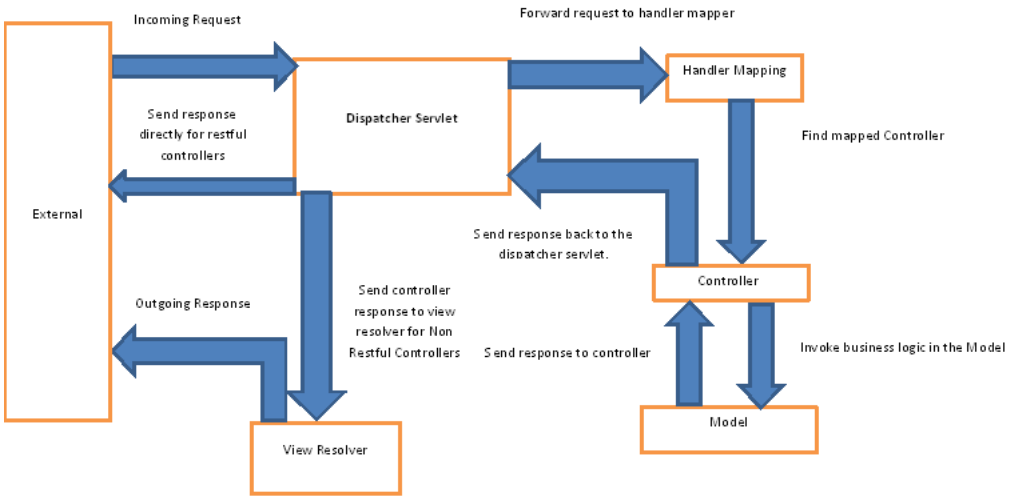


Fig 1 MVC Architecture flow

Как мы видим все действия происходят через один единственный DispatcherServlet.

Сконфигурировать наше Spring MVC-приложение мы можем с помощью Java-config, добавив зависимость spring-webmvc и установив над классом конфигурации @EnableWebMvc, которая применит дефолтные настройки - зарегистрирует некоторые специальные бины из Spring MVC и адаптирует их к нашим бинам. Но, если требуется тонкая настройка, то мы можем имплементировать интерфейс **WebMvcConfigurer** и переопределить необходимые методы.

Теперь нужно зарегистрировать конфигурацию в Spring Context это позволит сделать созданный нами класс MyWebAppInitializer, который нужно унаследовать от AbstractAnnotationConfigDispatcherServletInitializer, и передать в его методы классы нашей конфигурации RootConfig.class и App1Config.class:

```
public class MyWebAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[] { RootConfig.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { App1Config.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/*" };
    }
}
```

Своими внутренними методами он создает два экземпляра WebApplicationContext в виде объектов класса AnnotationConfigWebApplicationContext.

Если же у нас только один класс конфигурации, то его нужно передать в метод getRootConfigClasses(), а getServletConfigClasses() должен возвращать null.

25. Что такое ViewResolver?

Источники:
- Spring - View Resolution
- Javastudy - Spring MVC – описание интерфейса ViewResolver
- Baeldung - A Guide to the ViewResolver in Spring MVC

Все платформы MVC предоставляют способ работы с представлениями. Spring делает это с помощью **ViewResolver**, который позволяет отображать модели в браузере, не привязывая реализацию к определенной технологии представления.

ViewResolver сопоставляет имена представлений, возвращаемых методами контроллеров, с фактическими представлениями (html-файлами). Spring Framework поставляется с довольно большим количеством ViewResolver, например InternalResourceViewResolver, XmlViewResolver, ResourceBundleViewResolver и несколькими другими.

По умолчанию реализацией интерфейса ViewResolver является класс InternalResourceViewResolver.

Любым реализациям ViewResolver желательно поддерживать интернационализацию, то есть множество языков.

26. Расскажите про шаблон проектирования Front Controller, как он реализован в Spring?

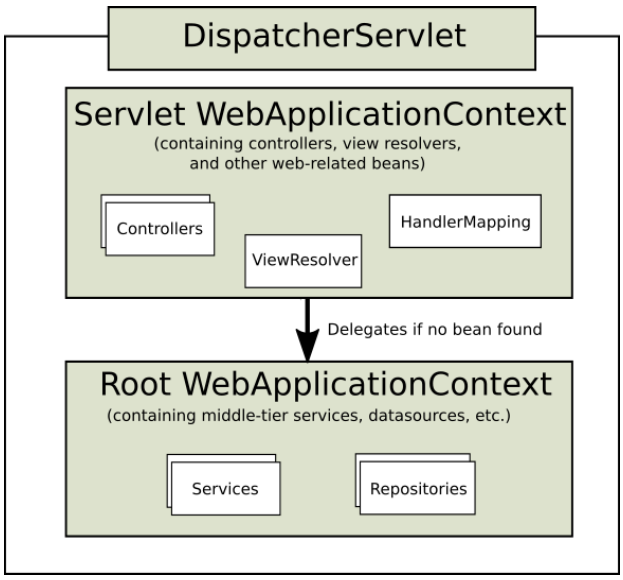
Источники:
Baeldung - A Guide to the Front Controller Pattern in Java
Howtodoinjava - ContextLoaderListener vs DispatcherServlet
Spring - DispatcherServlet
Spring - The DispatcherServlet

Паттерн Front Controller обеспечивает единую точку входа для всех входящих запросов. Все запросы обрабатываются одним фрагментом кода, который затем может делегировать ответственность за обработку запроса другим объектам приложения. Он также обеспечивает интерфейс для общего поведения, такого как безопасность, интернационализация и передача определенных представлений определенным пользователям.

В Spring в качестве Front Controller выступает DispatcherServlet, все действия проходят через него. Как правило в приложении задаётся только один DispatcherServlet с маппингом “/”, который перехватывает все запросы. Это и есть реализация паттерна Front Controller.

Однако иногда необходимо определить два и более DispatcherServlet-а, которые будут отвечать за свой собственный функционал. Например, чтобы один обрабатывал REST-запросы с маппингом “/api”, а другой обычные запросы с маппингом “/default”. Spring предоставляет нам такую возможность, и для начала нужно понять, что:

- Spring может иметь несколько контекстов одновременно. Одним из них будет корневой контекст, а все остальные контексты будут дочерними.
- Все дочерние контексты могут получить доступ к бинам, определенным в корневом контексте, но не наоборот. Корневой контекст не может получить доступ к бинам дочерних контекстов.
- Каждый дочерний контекст внутри себя может переопределить бины из корневого контекста.



Каждый DispatcherServlet имеет свой дочерний контекст приложения. DispatcherServlet по сути является сервлетом (он расширяет HttpServlet), основной целью которого является обработка входящих веб-запросов, соответствующих настроенному шаблону URL. Он принимает входящий URI и находит правильную комбинацию контроллера и вида. Веб-приложение может определять любое количество DispatcherServlet-ов. Каждый из них будет работать в своем собственном пространстве имен, загружая свой собственный дочерний WebApplicationContext (на рисунке - Servlet WebApplicationContext) с вьюшками, контроллерами и т.д. Например, когда нам нужно в одном Servlet WebApplicationContext определить обычные контроллеры, а в другом REST-контроллеры.

WebApplicationContext расширяет ApplicationContext (создаёт и управляет бинами и т.д.), но помимо этого он имеет дополнительный метод getServletContext(), через который у него есть возможность получать доступ к ServletContext-у.

ContextLoaderListener создает корневой контекст приложения (на рисунке - Root WebApplicationContext) и будет использоваться всеми дочерними контекстами, созданными всеми DispatcherServlet. Напомню, что корневой контекст приложения будет общим и может быть только один. Root WebApplicationContext содержит компоненты, которые видны всем дочерним контекстам, такие как сервисы, репозитории, компоненты инфраструктуры и т.д. После создания корневого контекста приложения он сохраняется в ServletContext как атрибут, имя которого:

```
WebApplicationContext.class.getName() + ".ROOT"
```

Чтобы из контроллера любого дочернего контекста обратиться к корневому контексту приложения, мы можем использовать класс WebApplicationContextUtils, содержащий статические методы:

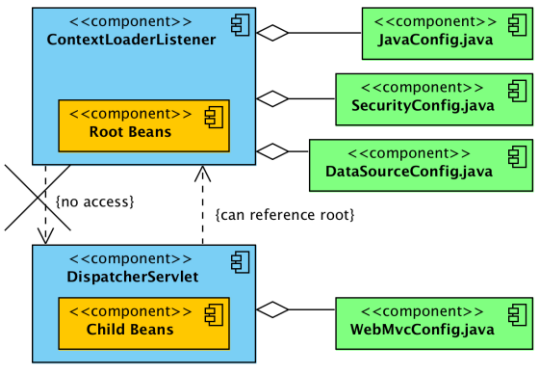
```
@Autowired
ServletContext context;

ApplicationContext ac =
    WebApplicationContextUtils.getWebApplicationContext(context);

if(ac == null){
    return "root application context is null";
}
```

ContextLoaderListener vs DispatcherServlet

1. ContextLoaderListener создает корневой контекст приложения.
2. Каждый DispatcherServlet создаёт себе один дочерний контекст.
3. Дочерние контексты могут обращаться к бинам, определенным в корневом контексте.
4. Бины в корневом контексте не могут получить доступ к бинам в дочерних контекстах (напрямую).
5. Все контексты добавляются в ServletContext.
6. Мы можем получить доступ к корневому контексту, используя класс WebApplicationContextUtils.



27. Чем отличаются Model, ModelMap и ModelAndView?

Источники:
- Baeldung - Model, ModelMap, and ModelView in Spring MVC
- Spring - Interface Model
- Spring - Class ModelMap
- Spring - Class ModelAndView

Model

Интерфейс, лежит в пакете spring-context. В методах контроллера мы можем использовать объекты Model для того, чтобы складывать туда данные, предназначенные для формирования представлений. Кроме того, в Model мы можем передать даже Map с атрибутами:

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

ModelMap

Этот класс наследуется от LinkedHashMap<String, Object> и по сути служит общим контейнером модели для Servlet MVC, но не привязан к нему, и лежит в пакете spring-context. Имеет все преимущества LinkedHashMap плюс несколько удобных методов:

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

ModelAndView

Этот класс лежит в пакете spring-webmvc и может одновременно хранить модели и представление, чтобы контроллер мог отдавать их в одном возвращаемом значении. Внутри содержит поле *private Object view*, куда записывает нужное представление, а также поле *private ModelMap model*, куда и складывает все атрибуты модели:

```
@GetMapping("/goToViewPage")
public ModelAndView passParametersWithModelAndView() {
    ModelAndView modelAndView = new ModelAndView("viewPage");
    modelAndView.addObject("message", "Baeldung");
    return modelAndView;
}
```

28. Расскажите про аннотации @Controller и @RestController. Чем они отличаются? Как вернуть ответ со своим статусом (например 213)?

Источники: Baeldung - @Controller and @RestController
Baeldung - Using Spring ResponseEntity

@Controller

@Controller помечает класс как контроллер HTTP-запросов. @Controller обычно используется в сочетании с аннотацией @RequestMapping, используемой в методах обработки запросов. Это просто дочерняя аннотация аннотации @Component и позволяет автоматически определять классы при сканировании пакетов.

@RestController

Аннотация @RestController была введена в Spring 4.0 для упрощения создания RESTful веб-сервисов. Это удобная аннотация, которая объединяет @Controller и @ResponseBody, что устраняет необходимость аннотировать каждый метод обработки запросов аннотацией @ResponseBody.

@ResponseBody сообщает контроллеру, что возвращаемый объект автоматически сериализуется в json или xml и передается обратно в объект HttpResponse. Контроллер использует Jackson message converter для конвертации входящих/исходящих данных. Как правило целевые данные представлены в json или xml.

ResponseEntity

Данный класс используется для формирования ответа HTTP с пользовательскими параметрами (заголовки, код статуса и тело ответа). ResponseEntity необходим, только если мы хотим кастомизировать ответ. Во всех остальных случаях достаточно использовать @ResponseBody.

Если мы хотим использовать ResponseEntity, то просто должны вернуть его из метода, Spring позаботится обо всем остальном.

```
@GetMapping("/customHeader")
ResponseEntity<String> customHeader() {

    HttpHeaders headers = new HttpHeaders();
    headers.add("Custom-Header", "foo");

    return new ResponseEntity<>(
      "Custom header set", headers, HttpStatus.OK);
}
```

Если клиент ждет от нас JSON/XML, мы можем параметризовать ResponseEntity конкретным классом и добавить к ответу заголовки и Http статус:

```
@RequestMapping(value = "/employees/{id}")
public ResponseEntity<EmployeeVO> getEmployeeById (@PathVariable("id") int id){
    if (id <= 3) {
```

```
        EmployeeVO employee =
            new EmployeeVO(1,"Lokesh","Gupta","howtodoinjava@gmail.com");
        return new ResponseEntity<EmployeeVO>(employee, HttpStatus.OK);
    }
    return new ResponseEntity(HttpStatus.NOT_FOUND);
    }
}
```

29. В чем разница между Filters, Listeners и Interceptors?

Источники:

Javadoc - Interface Filter
Javadoc - Interface ServletContextListener
Spring API - Interface Interceptor
Spring API - Interface HandlerInterceptor
O7planning - Руководство Spring MVC Interceptor
Baeldung - Introduction to Spring MVC HandlerInterceptor
Programmer - Filters, Interceptors, Listeners
Mkjava - Filter vs. Interceptor
Oracle - Servlet Filters and Event Listeners
Mkyong - ServletContextListener Example

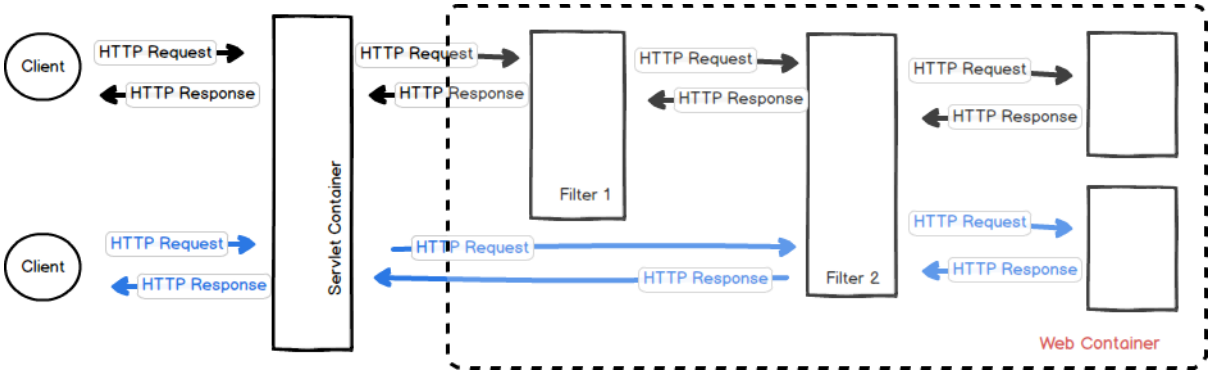
Filter

Это интерфейс из пакета javax.servlet, имплементации которого выполняют задачи фильтрации либо по пути запроса к ресурсу (сервлету, либо по статическому контенту), либо по пути ответа от ресурса, либо в обоих направлениях.

Фильтры выполняют фильтрацию в методе doFilter. Каждый фильтр имеет доступ к объекту FilterConfig, из которого он может получить параметры инициализации, и ссылку на ServletContext, который он может использовать, например, для загрузки ресурсов, необходимых для задач фильтрации. Фильтры настраиваются в дескрипторе развертывания веб-приложения.

В веб-приложении мы можем написать несколько фильтров, которые вместе называются цепочкой фильтров. Веб-сервер решает, какой фильтр вызывать первым, в соответствии с порядком регистрации фильтров.

Когда вызывается метод doFilter(ServletRequest request, ServletResponse response, FilterChain chain) первого фильтра, веб-сервер создает объект FilterChain, представляющий цепочку фильтров, и передаёт её в метод.

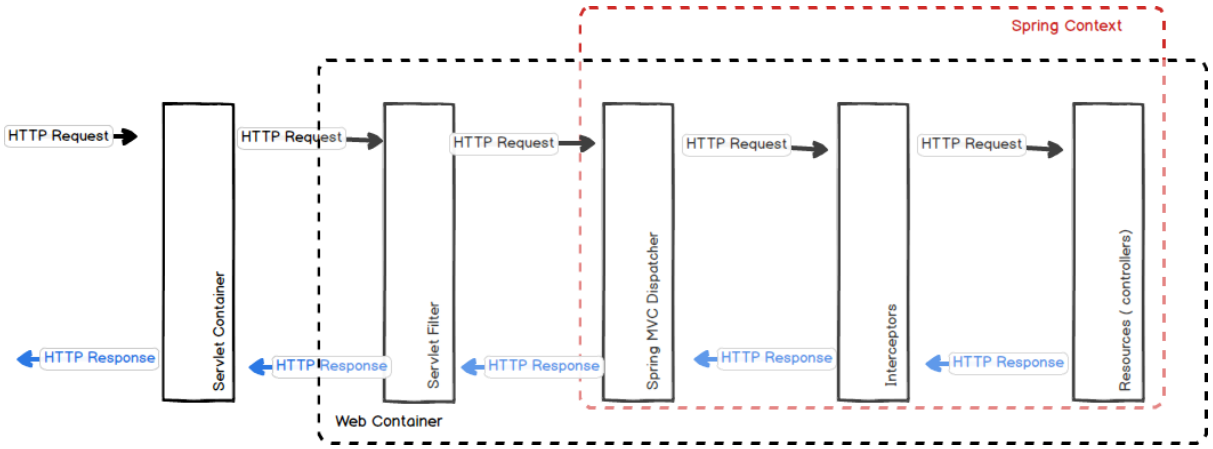


Interceptor

Это интерфейс из пакета org.aopalliance.intercept, предназначенный для аспектно-ориентированного программирования.

В Spring, когда запрос отправляется в Controller, перед тем как он в него попадёт, он может пройти через перехватчики **Interceptor** (0 или более). Это одна из реализаций АОП в Spring. Вы можете использовать Interceptor для выполнения таких задач, как запись в Log, добавление или обновление конфигурации перед тем, как запрос обработается Controller-ом.

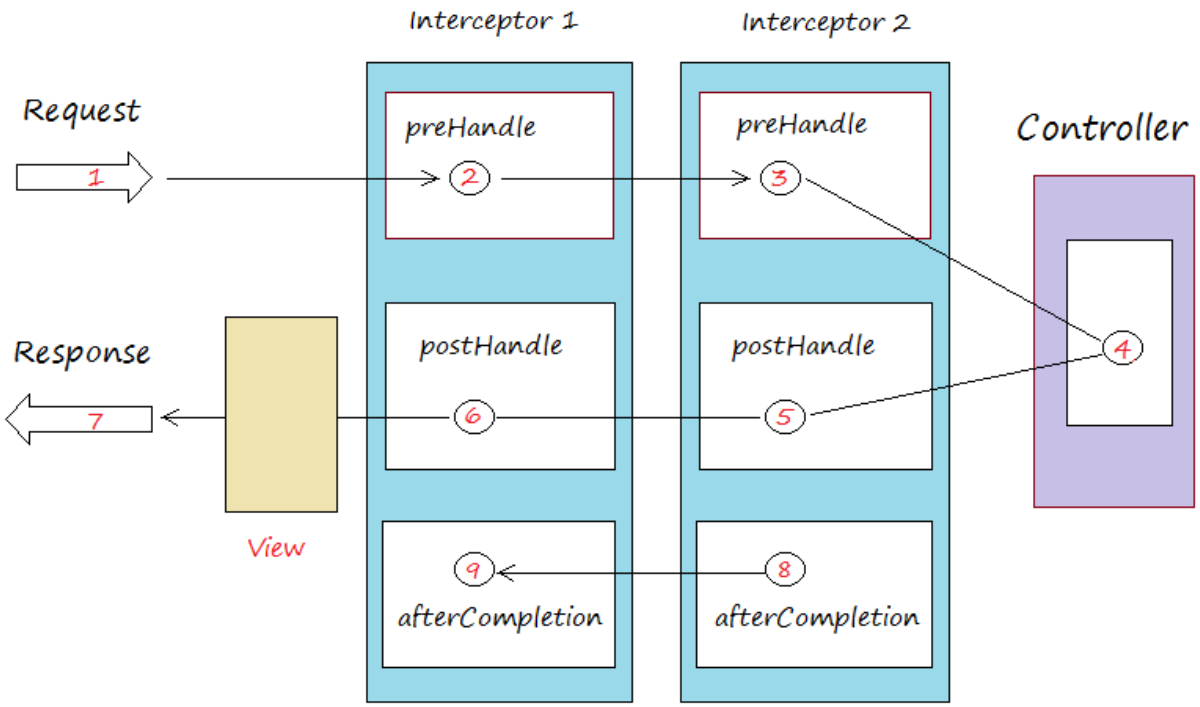
Стек перехватчиков: он предназначен для связывания перехватчиков в цепочку в определенном порядке. При доступе к перехваченному методу или полю перехватчик в цепочке перехватчиков вызывается в том порядке, в котором он был определен.



Мы можем использовать Interceptor-ы для выполнения логики до попадания в контроллер, после обработки в контроллере, а также после формирования представления. Также можем запретить выполнение метода контроллера. Мы можем указать любое количество перехватчиков.

Перехватчики работают с HandlerMapping и поэтому должны реализовывать интерфейс **HandlerInterceptor** или наследоваться от готового класса **HandlerInterceptorAdapter**. В случае реализации HandlerInterceptor нам нужно переопределить 3 метода, а в случае HandlerInterceptor, только необходимые нам:

- public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) - вызывается после того, как HandlerMapping определил соответствующий контроллер, но до того, как HandlerAdapter вызовет метод контроллера. С помощью этого метода каждый перехватчик может решить, прервать цепочку выполнения или направить запрос на испольнение дальше по цепочке перехватчиков до метода контроллера. Если этот метод возвращает true, то запрос отправляется следующему перехватчику или в контроллер. Если метод возвращает false, то исполнение запроса прекращается, обычно отправляя ошибку HTTP или записывая собственный ответ в response.
- public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, **ModelAndView modelAndView**) - отработает после контроллера, но перед формированием представления. Мы можем использовать этот метод для добавления дополнительных атрибутов в ModelAndView или для определения времени, затрачиваемого методом-обработчиком на обработку запроса клиента. Вы можете добавить больше объектов модели в представление, но вы не можете изменить HttpServletResponse, так как он уже зафиксирован.
- public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) - отработает после формирования представления. Вызывается только в том случае, если метод preHandle этого перехватчика успешно завершен и вернул true!



Следует знать, что HandlerInterceptor связан с бином **DefaultAnnotationHandlerMapping**, который отвечает за применение перехватчиков к любому классу, помеченному аннотацией @Controller.

Чтобы добавить наши перехватчики в конфигурацию Spring, нам нужно переопределить метод addInterceptors () внутри класса, который реализует WebMvcConfigurer:

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    // LogInterceptor applies to all URLs.
    registry.addInterceptor(new LogInterceptor());

    // This interceptor applies to URL /admin/oldLogin.
    // Using OldURLInterceptor to redirect to new URL.
    registry.addInterceptor(new OldLoginInterceptor())
        .addPathPatterns("/admin/oldLogin");

    // This interceptor applies to URLs like /admin/*
    // Exclude /admin/oldLogin
    registry.addInterceptor(new AdminInterceptor())
        .addPathPatterns("/admin/*")//
        .excludePathPatterns("/admin/oldLogin");
}
```

Filter vs. Interceptor

- Перехватчик основан на механизме Reflection, а фильтр основан на обратном вызове функции.
- Фильтр зависит от контейнера сервлета, тогда как перехватчик не зависит от него.
- Перехватчики могут работать только с запросами к контроллерам, в то время как фильтры могут работать почти со всеми запросами (например, js, .css и т.д.).

- ❖ Перехватчики в отличии от фильтров могут обращаться к объектам в контейнере Spring, что даёт им более изощренный функционал.

Порядок работы:

1. Фильтры до;
2. Перехватчики до;
3. Метод контроллера;
4. Перехватчики после;
5. Фильтры после.

HandlerInterceptor в основном похож на **Servlet Filter**, но в отличие от последнего он просто позволяет настраивать предварительную обработку с возможностью запретить выполнение самого обработчика и настраивать постобработку.

Согласно документации Spring, **фильтры** более мощные, например, они позволяют обмениваться объектами запроса и ответа, которые передаются по цепочке. Это означает, что фильтры работают больше в области запроса/ответа, в то время как HandlerInterceptors являются бинами и могут обращаться к другим компонентам в приложении. Обратите внимание, что фильтр настраивается в web.xml, а HandlerInterceptor в контексте приложения.

Java Listener

Listener (Слушатель) - это класс, который реализует интерфейс javax.servlet.ServletContextListener. Он инициализируется только один раз при запуске веб-приложения и уничтожается при остановке веб-приложения. Слушатель сидит и ждет, когда произойдет указанное событие, затем «перехватывает» событие и запускает собственное событие. Например, мы хотим инициализировать пул соединений с базой данных до запуска веб-приложения. ServletContextListener - это то, что нам нужно, он будет запускать наш код до запуска веб-приложения.

Все ServletContextListeners уведомляются об инициализации контекста до инициализации любых фильтров или сервлетов в веб-приложении.

Все ServletContextListeners уведомляются об уничтожении контекста после того, как все сервлеты и фильтры уничтожены.

Чтобы создать свой Listener нам достаточно создать класс, имплементирующий интерфейс ServletContextListener и поставить над ним аннотацию @WebListener:

```
@WebListener
public class MyAppServletContextListener
               implements ServletContextListener{

      //Run this before web application is started
      @Override
      public void contextInitialized(ServletContextEvent arg0) {
            System.out.println("ServletContextListener started");
      }

      @Override
      public void contextDestroyed(ServletContextEvent arg0) {
            System.out.println("ServletContextListener destroyed");
      }
```

```
}
```

30. Можно ли передать в GET-запросе один и тот же параметр несколько раз? Как?

Источники: Baeldung - Mapping a Multi-Value Parameter
Habr - Spring: вопросы к собеседованию

Да, можно принять все значения, используя массив в методе контроллера:

```
http://localhost:8080/login?name=Ranga&name=Ravi&name=Sathish
```

```
public String method(@RequestParam(value="name") String[] names){...}
```

или

```
http://localhost:8080/api/foos?id=1,2,3
----
IDs are [1,2,3]
```

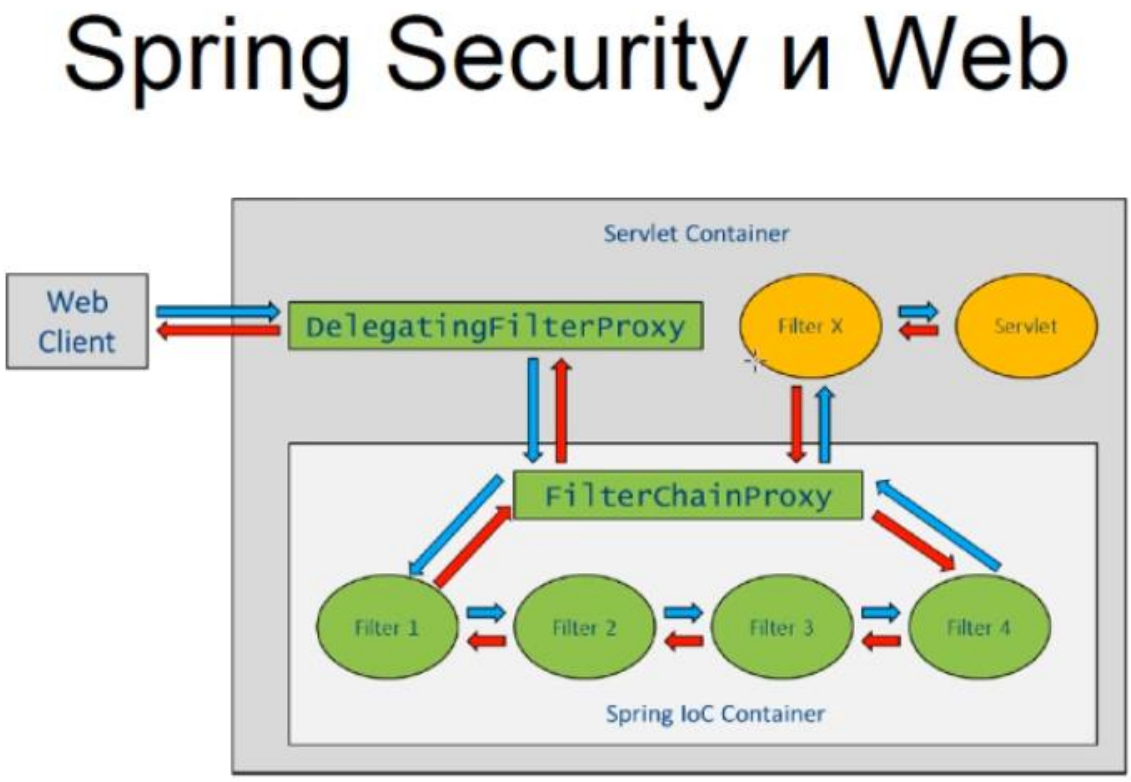
```
@GetMapping("/api/foos")
@ResponseBody
public String getFoos(@RequestParam List<String> id) {
    return "IDs are " + id;
}
```

31. Как работает Spring Security? Как сконфигурировать? Какие интерфейсы используются?

Источники:
Spring - Spring Security Architecture
Spring Security Documentation
Spring Security API
Wikibooks - Технический обзор Spring Security

Spring Security обеспечивает всестороннюю поддержку аутентификации, авторизации и защиты от распространенных эксплойтов. Он также обеспечивает интеграцию с другими библиотеками, чтобы упростить его использование.

Spring Security - это список фильтров в виде класса FilterChainProxy, интегрированного в контейнер сервлетов, и в котором есть поле List<SecurityFilterChain>. Каждый фильтр реализует какой-то механизм безопасности. Важна последовательность фильтров в цепочке.



Когда мы добавляем аннотацию @EnableWebSecurity добавляется DelegatingFilterProxy, его задача заключается в том, чтобы вызвать цепочку фильтров (FilterChainProxy) из Spring Security.

В Java-based конфигурации цепочка фильтров создается неявно.

Если мы хотим настроить свою цепочку фильтров, мы можем сделать это, создав класс, конфигурирующий наше Spring Security приложение, и имплементировав интерфейс WebSecurityConfigurerAdapter. В данном классе, мы можем переопределить метод:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
```

```
    http
        .csrf().disable()
        .authorizeRequests();
}
```

Именно этот метод конфигурирует цепочку фильтров Spring Security и логика, указанная в этом методе, настроит цепочку фильтров.

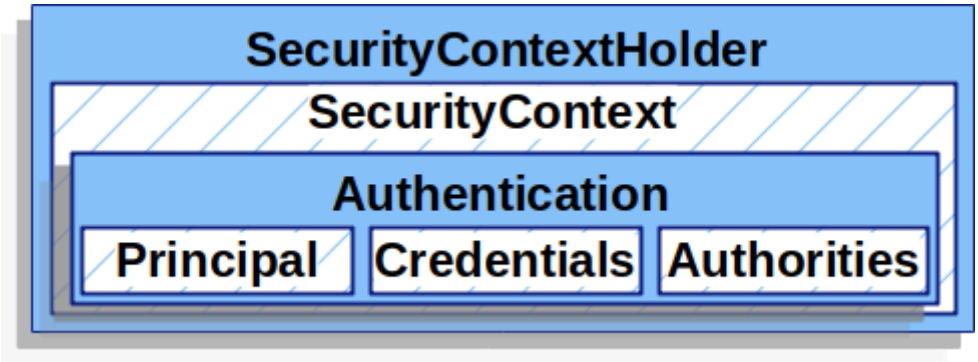
Основные классы и интерфейсы

SecurityContext - интерфейс, отражающий контекст безопасности для текущего потока. Является контейнером для объекта типа Authentication. (Аналог - ApplicationContext, в котором лежат бины).

По умолчанию на каждый поток создается один SecurityContext. SecurityContext-ы хранятся в SecurityContextHolder.

Имеет только два метода: getAuthentication() и setAuthentication(Authentication authentication).

SecurityContextHolder - это место, где Spring Security хранит информацию о том, кто аутентифицирован. Класс, хранящий в ThreadLocal SecurityContext-ы для каждого потока, и содержащий статические методы для работы с SecurityContext-ами, а через них с текущим объектом Authentication, привязанным к нашему веб-запросу.



Authentication - объект, отражающий информацию о текущем пользователе и его привилегиях. Вся работа Spring Security будет заключаться в том, что различные фильтры и обработчики будут брать и класть объект Authentication для каждого посетителя. Кстати объект Authentication можно достать в Spring MVC контроллере командой `SecurityContextHolder.getContext().getAuthentication()`. Authentication имеет реализацию по умолчанию - класс UsernamePasswordAuthenticationToken, предназначенный для хранения логина, пароля и коллекции Authorities.

Principal - интерфейс из пакета java.security, отражающий учетную запись пользователя. В терминах логин-пароль это логин. В интерфейсе Authentication есть метод getPrincipal(), возвращающий Object. При аутентификации с использованием имени пользователя/пароля Principal реализуется объектом типа UserDetails.

Credentials - любой Object; то, что подтверждает учетную запись пользователя, как правило пароль (отпечатки пальцев, пин - всё это Credentials, а владелец отпечатков и пина - Principal).

GrantedAuthority - полномочия, предоставленные пользователю, например, роли или уровни доступа.

UserDetails - интерфейс, представляющий учетную запись пользователя. Как правило модель нашего пользователя должна имплементировать его. Она просто хранит

пользовательскую информацию в виде логина, пароля и флагов isAccountNonExpired, isAccountNonLocked, isCredentialsNonExpired, isEnabled, а также коллекции прав (ролей) пользователя. Данная информация позже инкапсулируется в объекты Authentication.

UserDetailsService - интерфейс объекта, реализующего загрузку пользовательских данных из хранилища. Созданный нами объект с этим интерфейсом должен обращаться к БД и получать оттуда юзеров.

AuthenticationManager - основной стратегический интерфейс для аутентификации. Имеет только один метод, который срабатывает, когда пользователь пытается аутентифицироваться в системе:

```
public interface AuthenticationManager {
  Authentication authenticate(Authentication authentication)
    throws AuthenticationException;
}
```

AuthenticationManager может сделать одну из 3 вещей в своем методе authenticate():

1. вернуть Authentication (с authenticated=true), если предполагается, что вход осуществляет корректный пользователь.
2. бросить AuthenticationException, если предполагается, что вход осуществляет некорректный пользователь.
3. вернуть null, если принять решение не представляется возможным.

Наиболее часто используемая реализация AuthenticationManager - родной класс ProviderManager, который содержит поле `private List<AuthenticationProvider> providers` со списком AuthenticationProvider-ов и итерирует запрос аутентификации по этому списку AuthenticationProvider-ов. Идея такого разделения - поддержка различных механизмов аутентификации на сайтах.

AuthenticationProvider - интерфейс объекта, выполняющего аутентификацию. Имеет массу готовых реализаций. Также можем задать свой тип аутентификации. Как правило в небольших проектах одна логика аутентификации - по логину и паролю. В проектах побольше логик может быть несколько: Google-аутентификация и т.д., и для каждой из них создается свой объект AuthenticationProvider.

AuthenticationProvider немного похож на AuthenticationManager, но у него есть дополнительный метод, позволяющий вызывающей стороне спрашивать, поддерживает ли он переданный ему объект Authentication, возможно этот AuthenticationProvider может поддерживать только аутентификацию по логину и паролю, но не поддерживать Google-аутентификацию:

```
boolean supports(java.lang.Class<?> authentication)
```

PasswordEncoder - интерфейс для шифрования/расшифровывания паролей. Одна из популярных реализаций - BCryptPasswordEncoder.

В случае, если нам необходимо добавить логику при успешной/неудачной аутентификации, мы можем создать класс и имплементировать интерфейсы AuthenticationSuccessHandler и AuthenticationFailureHandler соответственно, переопределив их методы.

Как это работает с формой логина и UserDetailsService:

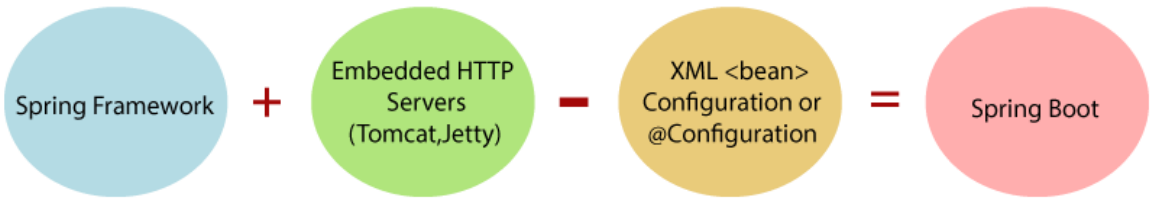
- Пользователь вводит в форму и отправляет логин и пароль.

- UsernamePasswordAuthenticationFilter создает объект Authentication - UsernamePasswordAuthenticationToken, где в качестве Principal - логин, а в качестве Credentials - пароль.
- Затем UsernamePasswordAuthenticationToken передаёт объект Authentication с логином и паролем AuthenticationManager-у.
- AuthenticationManager в виде конкретного класса ProviderManager внутри своего списка объектов AuthenticationProvider, имеющих разные логики аутентификации, пытается аутентифицировать посетителя, вызывая его метод authenticate(). У каждого AuthenticationProvider-а:
  - Метод authenticate() принимает в качестве аргумента незаполненный объект Authentication, например только с логином и паролем, полученными в форме логина на сайте. Затем с помощью UserDetailsService метод идёт в БД и ищет такого пользователя.
  - Если такой пользователь есть в БД, AuthenticationProvider получает его из базы в виде объекта UserDetails. Объект Authentication заполняется данными из UserDetails - в него включаются Authorities, а в Principal записывается сам объект UserDetails, содержащий пользователя.
  - Затем этот метод возвращает заполненный объект Authentication **(прошли аутентификацию)**. Вызывается AuthenticationSuccessHandler.
  - Если логин либо пароль неверные, то выбрасывается исключение. Вызывается AuthenticationFailureHandler.
- Затем этот объект Authentication передается в AccessDecisionManager и получаем решение на получение доступа к запрашиваемой странице **(проходим авторизацию)**.

32. Что такое Spring Boot? Какие у него преимущества? Как конфигурируется? Подробно

Источники:
- Habr - Обратная сторона Spring / Spring Boot
- Habr - Введение в Spring Boot
- Javatpoint - Learn Spring Boot Tutorial
- Spring Boot Reference Documentation
- Spring Boot - Getting Started

Spring Boot - это модуль Spring-а, который предоставляет функцию RAD для среды Spring (Rapid Application Development - Быстрая разработка приложений). Он обеспечивает более простой и быстрый способ настройки и запуска как обычных, так и веб-приложений. Он просматривает наши пути к классам и настроенные нами бины, делает разумные предположения о том, чего нам не хватает, и добавляет эти элементы.



Spring Boot представляет собой комбинацию Spring Framework и встроенного контейнера сервлетов и отсутствие (или минимальное наличие) конфигурации приложения.

Ключевые особенности и преимущества Spring Boot:

1. Простота управления зависимостями (spring-boot-starter-* в pom.xml).

   Чтобы ускорить процесс управления зависимостями Spring Boot неявно упаковывает необходимые сторонние зависимости для каждого типа приложения на основе Spring и предоставляет их разработчику в виде так называемых starter-пакетов.

   Starter-пакеты представляют собой набор удобных дескрипторов зависимостей, которые можно включить в свое приложение. Это позволяет получить универсальное решение для всех технологий, связанных со Spring, избавляя программиста от лишнего поиска необходимых зависимостей, библиотек и решения вопросов, связанных с конфликтом версий различных библиотек.

   Например, если вы хотите начать использовать Spring Data JPA для доступа к базе данных, просто включите в свой проект зависимость spring-boot-starter-data-jpa (вам не придется искать совместимые драйверы баз данных и библиотеки Hibernate). Если вы хотите создать Spring web-приложение, просто добавьте зависимость spring-boot-starter-web, которая подтянет в проект все библиотеки, необходимые для разработки Spring MVC-приложений, таких как spring-webmvc, jackson-json, validation-api и Tomcat.

   Другими словами, Spring Boot собирает все общие зависимости и определяет их в одном месте, что позволяет разработчикам просто их использовать. Также при использовании Spring Boot, файл pom.xml содержит намного меньше строк, чем в Spring-приложениях.

2. Автоматическая конфигурация.

   Автоматическая конфигурация включается аннотацией @EnableAutoConfiguration. (входит в состав аннотации @SpringBootApplication)

   После выбора необходимых для приложения starter-пакетов Spring Boot попытается автоматически настроить Spring-приложение на основе выбранных jar-зависимостей, доступных в classpath классов, свойств в application.properties и т.п. Например, если добавим spring-boot-starter-web, то Spring boot автоматически сконфигурирует такие бины как DispatcherServlet, ResourceHandlers, MessageSource итд

   Автоматическая конфигурация работает в последнюю очередь, после регистрации пользовательских бинов и всегда отдает им приоритет. Если ваш код уже зарегистрировал бин DataSource — автоконфигурация не будет его переопределять.

3. Встроенная поддержка сервера приложений/контейнера сервлетов (Tomcat, Jetty, итд). Каждое Spring Boot web-приложение включает встроенный web-сервер. Не нужно беспокоиться о настройке контейнера сервлетов и развертывания приложения в нем. Теперь приложение может запускаться само как исполняемый .jar-файл с использованием встроенного сервера.

4. Готовые к работе функции, такие как метрики, проверки работоспособности, security и внешняя конфигурация.

5. Инструмент CLI (command-line interface) для разработки и тестирования приложения Spring Boot.

6. Минимизация boilerplate кода (код, который должен быть включен во многих местах практически без изменений), конфигурации XML и аннотаций.

Как происходит автоконфигурация в Spring Boot:

1. Отмечаем main класс аннотацией @SpringBootApplication (аннотация инкапсулирует в себе: @SpringBootConfiguration, @ComponentScan, @EnableAutoConfiguration), таким образом наличие @SpringBootApplication включает сканирование компонентов, автоконфигурацию и показывает разным компонентам Spring (например, интеграционным тестам), что это Spring Boot приложение.

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

2. @EnableAutoConfiguration импортирует класс EnableAutoConfigurationImportSelector. Этот класс не объявляет бины сам, а использует так называемые фабрики.

3. Класс EnableAutoConfigurationImportSelector смотрит в файл META-INF/spring.factories и загружает оттуда список значений, которые являются именами классов (авто)конфигураций, которые Spring Boot импортирует. Т.е. аннотация @EnableAutoConfiguration просто импортирует ВСЕ (более 150) перечисленные в spring.factories конфигурации, чтобы предоставить нужные бины в контекст приложения.

4. Каждая из этих конфигураций пытается сконфигурировать различные аспекты приложения (web, JPA, AMQP и т.д.), регистрируя нужные бины. Логика при регистрации бинов управляется набором @ConditionalOn* аннотаций. Можно указать, чтобы бин создавался при наличии класса в classpath (@ConditionalOnClass), наличии существующего бина (@ConditionalOnBean), отсутствии бина (@ConditionalOnMissingBean) и т.п. Таким образом наличие конфигурации не значит, что бин будет создан, и в большинстве случаев конфигурация ничего делать и создавать не будет.

5. Созданный в итоге AnnotationConfigEmbeddedWebApplicationContext ищет в том же DI контейнере фабрику для запуска embedded servlet container.

6. Servlet container запускается, приложение готово к работе!

33. Расскажите про нововведения Spring 5

Источники: Spring - What's New in Spring Framework 5.x
Shell26 - Новое в Spring 5

- Используется **JDK 8**+ (Optional, CompletableFuture, Time API, java.util.function, default methods)
- Поддержка Java 9 (Automatic-Module-Name in 5.0, module-info in 6.0+, ASM 6)
- Поддержка **HTTP/2** (TLS, Push), **NIO/NIO.2**
- Поддержка **Kotlin**
- **Реактивность** (веб-инфраструктура с реактивным стеком, «Spring WebFlux»)
- Null-safety аннотации(**@Nullable**), новая документация
- Совместимость с **Java EE 8** (Servlet 4.0, Bean Validation 2.0, JPA 2.2, JSON Binding API 1.0)
- Поддержка **JUnit 5** + Testing Improvements (conditional and concurrent)
- Удалена поддержка: Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava