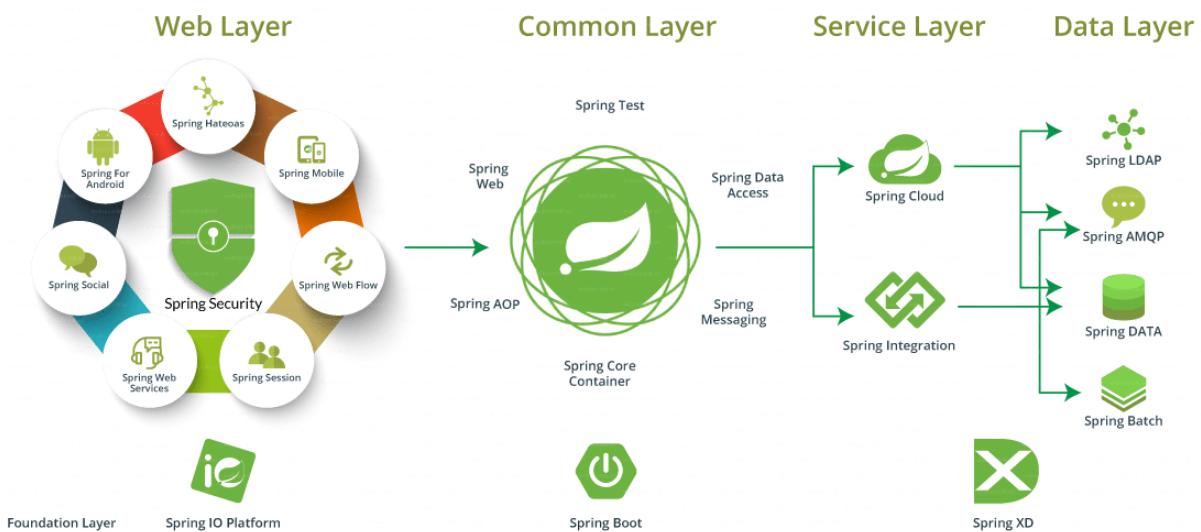


## SPRING Framework &amp; Ecosystem

edureka!

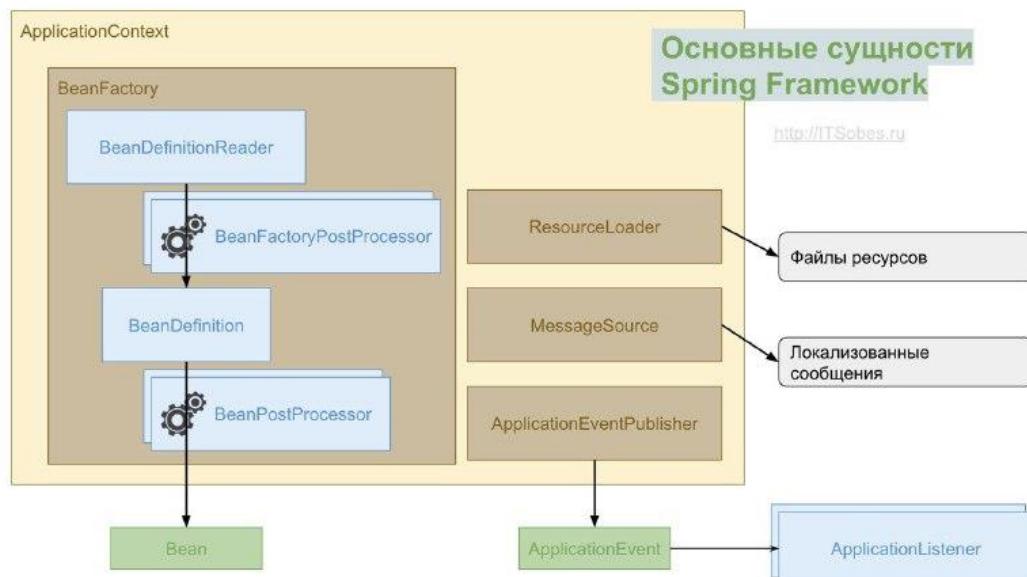


Универсальный фреймворк с открытым исходным кодом для Java с 2002 года.

Позволяет создавать масштабируемые и легко поддерживаемые решения, что сильно ускоряет разработку. Высокая модульность: каждый модуль способен запускаться как самостоятельное приложение, либо группой (фреймворк во фреймворке).

- Функционал MVC
- Работа с базами данных Spring Data
- Секьюрность (безопасность) Spring Security
- Обмен сообщениями
- Шаблонизаторы
- Множество готовых решений, модулей (Core, AOP, Web, Reactive)
- Удобство и простота использования (Spring Boot)
- Отлично подходит для микросервисной архитектуры (Actuator, Cloud)
- Support, Community

Из каких основных сущностей состоит Spring-приложение?



**Bean** – объект бизнес-логики в терминологии Spring Framework.

**BeanDefinition** – декларация, описание того, как создавать бин. Объект хранит его тип, метаинформацию, набор параметров для конструктора.

**BeanFactory** – главная точка входа в DI-контейнер. Хранит BeanDefinition-ы, умеет создавать по ним экземпляры бинов, или выдавать существующие, в зависимости от области видимости Scope.

**BeanPostProcessor** – до-настраивает только что созданные бины, ПЕРЕД тем как положить их в контейнер. Типичное место, чтобы оборачивать бины в прокси.

Также с помощью такого постпроцессора внедряются `@Autowired` зависимости. Постпроцессоры бинов живут внутри экземпляра `BeanFactory`.

**BeanFactoryPostProcessor** – тоже пост-обработчик, но для деклараций бинов (BeanDefinition). Обычно используется для модификации параметров или класса, из которых будут строиться бины.

Для создания определений бинов в основном применяются классы и интерфейсы `*BeanDefinitionReader`. Некоторые из них вызываются прямо из контекста приложения, другие реализуют `BeanFactoryPostProcessor`. Один такой пост-процессор, например, отвечает за добавление определений бинов по аннотациям `@Component` и `@Configuration`.

Реализация **интерфейса ApplicationContext** – основное хранилище конфигурации Spring-приложения (или его части). Контекст неизменяем, но может быть целиком перезагружен. Xml-файл конфигурации на старте приложения превращается в объект `*XmlApplicationContext`.

Для конфигурации на аннотациях создается `AnnotationConfigApplicationContext`. Контекст выполняет четыре разных обязанности:

**1. DI-контейнер.** `ApplicationContext` функционирует как специальная реализация `BeanFactory`. Он также производит и хранит бины, но, в отличие от обычных фабрик, контексты в приложении составляют иерархию. Определения бинов из дочерних контекстов перекрывают родительские.

**2. Загрузка ресурсов.** Под интерфейсом `ResourceLoader` контекст занимается загрузкой в память приложения файлов, как из classpath, так и из остальной файловой системы.

**3. Публикация событий приложения.** Контекст распространяет в приложении «события» – наследники `ApplicationEvent`. **Любой бин**, которому нужно получать уведомления об этих событиях, просто реализует интерфейс `ApplicationListener`.

Таким образом реализуется **паттерн наблюдатель**.

**4. Интернационализация.** По коду, набору аргументов и локали, через интерфейс контекста `MessageSource` можно получать локализованные текстовые сообщения для пользователей.

## Что такое инверсия контроля (IoC) и внедрение зависимостей (DI)?

Dependency Injection (DI) – это ЧАСТЬ Inversion of Control (IoC), не взаимозаменяемы!

IoC – аутсорсинг создания и управления объектами. Т.е. передача программистом прав на создание и управление объектами Spring-у.

DI – аутсорсинг добавления/внедрения зависимостей. DI делает объекты нашего приложения слабо зависимыми друг от друга.

**Инверсия контроля (inversion of control, IoC)** – принцип проектирования, по которому контроль над потоком управления передается фреймворку. Управляющий и прикладной код разделяются. При разработке модуля этот подход избавляет от необходимости знать о других модулях программы и деталях их взаимодействия. Такой код становится более переиспользуемым и модульным, уменьшает связность.

**Внедрение зависимостей (Dependency Injection, DI)** – одна из реализаций IoC.

При взаимодействии с другими модулями, программа оперирует высокогоровневыми абстракциями, тогда как конкретная её реализация поставляется фреймворком.

**Стандартная реализация DI** – фреймворк инстанцирует все сервисы, и складывает их в IoC-контейнер. При этом специальная сущность, **Service Locator**, занимается поиском соответствия реализаций абстракциям и их внедрением.

Как эти принципы реализованы в Spring?

Spring – набор различных библиотек. DI реализуется одной из основных – **Spring IoC**.

Сущности бизнес-логики в Spring, как и в JavaEE называются **beans**.

Бины объявляются различными способами, корни большинства из них лежат в понятии Configuration. В качестве контейнера бинов выступает **ApplicationContext**.

Чтобы передать инициализацию зависимости контексту, она помечается аннотацией `@Autowired`.

### Inversion of Control

Способы конфигурации Spring Container:

- XML file (устаревший способ)
- Annotations + XML file (современный способ)
- Java code (современный способ)

### Dependency Injection

DI – аутсорсинг добавления/внедрения зависимостей. DI делает объекты нашего приложения слабо зависимыми друг от друга.

Способы внедрения зависимостей:

- С помощью конструктора
- С помощью сеттеров
- Autowiring

DI с помощью конструктора:

```
<bean id = "myPet"
      class = "ioc.Cat">
</bean>
```

• constructor-arg – аргумент конструктора

• ref – ссылка на bean id

За кулисами:

```
Cat myPet = new Cat();
```

```
<bean id = "myPerson"
      class="ioc.Person">
    <constructor-arg ref="myPet"/>
</bean>
```

Person myPerson = new Person(myPet);

DI с помощью сеттера:

```
<bean id = "myPet"
      class = "ioc.Cat">
</bean>
```

Первая буква в слове «ref» становится заглавной и в начало слова добавляется «set». После чего вызывается получившийся метод.

```
Person myPerson = new Person();
myPerson.setPet(myPet);
```

За кулисами:

```
Cat myPet = new Cat();
```

Person myPerson = new Person();

myPerson.setPet(myPet);

## Что такое IoC контейнер?

В Spring Framework контейнер отвечает за создание, настройку и сборку объектов, известных как бины, а также за управление их жизненным циклом.

Контейнер представлен интерфейсом **ApplicationContext**.

Контейнер IoC «под капотом» является **ассоциативным массивом** (набором пар элементов), где ключом является String идентификатор, а значением является объект. Этот объект вызывается бином (т.е. создаётся IoC с помощью Reflection API вызовом конструктора бина с последующим внедрением зависимостей).

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:** метаданныечитываются из файла с расширением .xml

Атрибут **id** представлен **String, строкой**, которая идентифицирует бин-компонент. Атрибут **class** определяет **ТИП bean-компонента** и использует полное имя класса.



- **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 через аннотации + XML-файл, то настройки XML переопределят настройки аннотаций.

- **Java-код начиная со Spring 3.0**, используя Java-код, а не файлы XML, можно определять настройки в специальном классе, помеченном аннотацией `@Configuration`. Появились аннотации `@Configuration`, `@Bean`, `@Import` и `@DependsOn` и т.д.

Этапы инициализации контекста с примерами тут: <https://habr.com/ru/post/222579/>

## Что такое Bean в спринге?

Bean – это объект, управляемый спрингом.

В Spring – все бины (и сервисы, и DAO, и контроллеры (группы сервлетов)).

Spring Bean (или просто bean) – это объект, который создаётся и управляет Spring Container

ApplicationContext представляет собой Spring Container. Поэтому для получения бина из Spring Container нам нужно создать ApplicationContext.

```
ClassPathXmlApplicationContext context =  
    new ClassPathXmlApplicationContext( configLocation: "applicationContext.xml" );  
  
Pet pet = context.getBean( name: "myPet", Pet.class );
```

### Расскажите про аннотацию @Bean?

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

Бин — это объект, который создается, собирается и управляется контейнером Spring IoC.

Бины и их зависимости отражаются в **методах конфигурации**, используемых контейнером.

**@Bean** – это аннотация Spring Framework, она используется **НАД МЕТОДОМ** для указания того, что этот метод создает, настраивает и инициализирует новый объект, управляемый Spring IoC контейнером.

Методы с аннотацией **@Bean** 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, т.к. 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 «на лету» после фазы компиляции программы.

Hibernate использует cglib для генерации динамических прокси. Например, он не вернет полный объект, хранящийся в БД, 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.

### Расскажите про аннотацию `@Component`?

`@Component` – простой способ сделать объявление класса **объявлением Spring-бина**. Этой аннотацией помечаем КЛАСС, если хотим, чтобы из этого класса был создан бин.

Из всех компонентов, которые попали в сканирование (о которых знает `@ComponentScan`), будут созданы **декларации бинов Bean Definition**.

Аннотации `@Service`, `@Repository` и `@Controller` – это **алиасы** аннотации `@Component`. Сами по себе они не добавляют поведения, и технически в рамках ядра Spring Framework работают так же.

Эти аннотации называют «Stereotype annotations». Их главное отличие – семантика, логическая роль компонентов:

`@Service` – реализация бизнес-логики

`@Repository` – хранилище данных: «репозиторий» из Domain-Driven Design или DAO;

`@Controller` – обработка веб-запросов (методы `@RequestMapping`)  
Сторонние компоненты могут пользоваться этой семантикой. Например, трансляция исключений Persistence API работает именно на компонентах стереотипа `@Repository`.

В отдельных случаях кроме семантики может меняться и поведение кода библиотек.

**Для маппинга (связывания) с URL для всего класса или для конкретного метода** обработчика используется аннотация `@RequestMapping`.

### Чем отличаются аннотации `@Bean` и `@Component`?

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

`@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**
- **возможно объявить бин независимо от объявления класса**, что позволяет использовать классы из сторонних библиотек, у которых не можем указать аннотацию `@Component`
- с аннотацией `@Bean` можно настроить `initMethod`, `destroyMethod`, `autowireCandidate`, делая **создание бина более гибким**

### Расскажите про аннотации `@Service` и `@Repository`. Чем они отличаются?

`@Service` и `@Repository` являются частными случаями аннотации `@Component`. Технически они одинаковы, но используются их для разных целей.

`@Service` указывает, что используем **сервис-слой, отвечающий за бизнес-логику**. Аннотация Spring `@Repository` указывает на то, что класс **предоставляет механизм для CRUD операций**: для хранения, извлечения, поиска, обновления и удаления операций с объектами. Этот слой отвечает за доступ к данным.

Абстракция `@Repository` Spring Data позволяет значительно сократить объем шаблонного кода для реализации **уровней доступа к данным** для различных хранилищ сохраняемости.

Задача `@Repository` заключается также в том, чтобы отлавливать определенные исключения персистентности (Persistence Exception) и прорасывать их **как одно непроверяемое исключение** Spring Framework. Для этого в контекст должен быть добавлен класс PersistenceExceptionTranslationPostProcessor.

### Расскажите про аннотацию `@Autowired`

Этой аннотацией помечают **конструктор, поле, сеттер-метод или метод конфигурации**, сигнализируя, что им обязательно требуется внедрение зависимостей.

```
@Target  ({ CONSTRUCTOR , METHOD , PARAMETER , FIELD , ANNOTATION_TYPE })
@Retention ( RUNTIME )
@Documented
public @interface Autowired
```

```
@Component ("catBean")
public class Cat implements Pet {
```

#### Constructor injection

```
@Component ("personBean")
public class Person {
    private Pet pet;

    @Autowired
    public Person(Pet pet) {
        this.pet = pet;
    }
}
```

#### Setter injection

```
@Autowired
public void setPet(Pet pet) {
    this.pet = pet;
}
```

#### Field injection

```
@Autowired
private Pet pet;
```

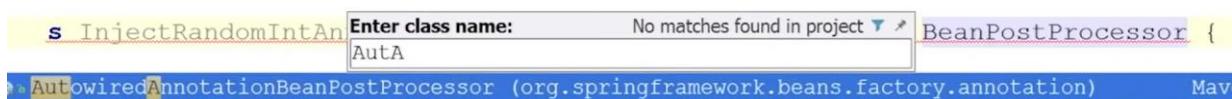
#### Any method injection

```
@Autowired
public void anyMethodName(Pet pet) {
    this.pet = pet;
}
```

За обработку `@Autowired` отвечает **BeanPostProcessor** (соответствует этапу ЖЦ бина).

То есть фактическое внедрение выполняется через BeanPostProcessor, что, в свою очередь, означает, что нельзя использовать `@Autowired` для внедрения ссылок в типы BeanPostProcessor или BeanFactoryPostProcessor.

По умолчанию проверяет наличие аннотации `@Autowired` класс `AutowiredAnnotationBeanPostProcessor`.



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.

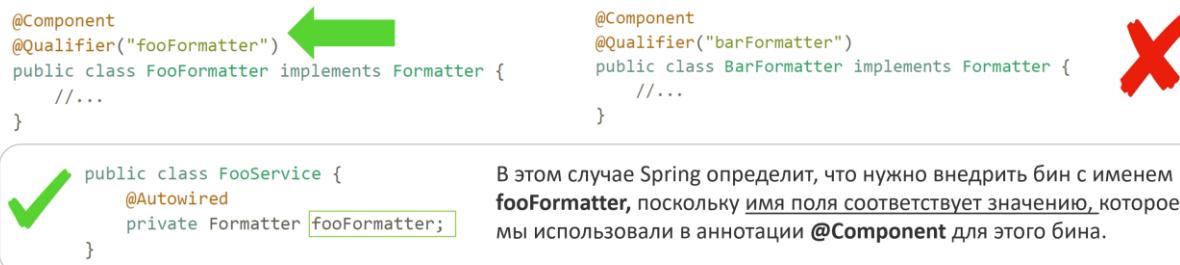
Аннотация **@Autowired** также принимает необязательный логический аргумент с именем **required**. По умолчанию его значение равно `true`. Можно явно установить для него значение `false`, для которого Spring не будет генерировать исключение при сбое автоматического подключения: `@Autowired(required = false)`.

По умолчанию Spring распознает объекты для вставки по ТИПУ. Если в контейнере доступно **более одного бина одного и того же типа**, будет выброшено исключение.

Для избежания этого можно указать аннотацию Spring Framework **@Qualifier** ("название поля"), где название `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.



### Расскажите про аннотацию `@Resource`

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 `@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 `@Resource` не нужно

Расскажите про аннотацию `@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 подбрана 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` (см. ниже)

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

Расскажите про аннотацию `@Lookup`

Метод, аннотированный `@Lookup`, сообщает Spring, чтобы он возвращал экземпляр типа возвращаемого значения метода, когда мы его вызываем.

По сути, Spring переопределит аннотированный метод и будет использовать тип возвращаемого значения и параметры метода в качестве аргументов для BeanFactory `#getBean`.

Обычно бины в приложении Spring являются синглтонами, и для внедрения зависимостей используют конструктор или сеттер.

Но бывает и другая ситуация: имеется бин *Car* — singleton bean, и ему требуется каждый раз новый экземпляр бина *Passenger* (каршеринг).

То есть *Car* — singleton, а *Passenger* — так называемый прототип-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**.

Создаётся метод-заглушка в бине *Car* и помечается специальным образом — аннотацией `@Lookup`. Этот метод должен возвращать бин *Passenger*, **каждый раз новый**.

Контейнер Spring под капотом создаст подкласс и переопределит этот метод и будет выдавать новый экземпляр бина *Passenger* при каждом вызове аннотированного метода. Даже если в заглушке он возвращает null (а так и надо делать, все равно этот метод будет переопределен).

### Пример использования:

Создадим класс *Car*, singleton, и сделаем его бином с помощью аннотации `@Component`:

```
1. @Component           Допустим, в бине есть метод drive(), и при каждом вызове метода drive() бину Car требуется новый
2. public class Car {   экземпляр бина Passenger — сегодня пассажир Петя, завтра — Вася. То есть бин Passenger прототипный.
3.
4.     @Lookup           Для получения этого бина надо написать метод-заглушку createPassenger().
5.     public Passenger createPassenger() {   аннотировать его с помощью @Lookup:
6.         return null;
7.     }                   Контейнер Spring переопределит этот метод-заглушку и будет выдавать при его вызове каждый раз новый
8.                           экземпляр Passenger.
9.     public String drive(String name) {
10.         Passenger passenger = createPassenger();
11.         passenger.setName(name);
12.         return "car with " + passenger.getName();
13.     }
14. }
```

```
1. @Component
2. @Scope("prototype")
3. public class Passenger {
4.     private String name;
5.
6.     public String getName() {
7.         return name;
8.     }
9.
10.    public void setName(String name) {
11.        this.name = name;
12.    }
13. }
```

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

Внедрение зависимости с помощью `Lookup` метода в коде встречается нечасто, но это хороший способ **внедрить бин-прототип в бин-singleton**.

Пример и код тут: <https://sysout.ru/kak-ispolzovat-annotsiyu-lookup/>

## \*Как работает инъекция прототипа в синглтон?

Вопрос касается также различия области видимости (скоупов) singleton и prototype в Spring Framework. Допустим ситуацию, когда в singleton-компонент внедряется зависимость со scope prototype – когда будет создан её объект?

Если просто добавить к определению бина аннотацию `@Scope(SCOPE_PROTOTYPE)`, и использовать этот бин в синглтоне через аннотацию `@Autowired` – будет создан только один объект. Потому что синглтон создается только однажды, и обращение к прототипу случится тоже однажды при его создании (при внедрении зависимости).

Примитивный способ получать новый объект при каждом обращении – отказаться от `@Autowired`, и доставать его из контекста вручную, вызывая `context.getBean(MyPrototype.class)`

Воспользоваться автоматическим внедрением зависимостей можно через **внедрение метода** (паттерн «Команда»). Внедряется не сам объект, а производящий его метод.

**Команда (Command)** — поведенческий шаблон проектирования, используемый при ООП, для обработки команды в виде объекта и представляющий ДЕЙСТВИЕ. То есть Объект команды заключает в себе само действие и его параметры.

**Более красивый декларативный способ – правильно настроить определение бина.** В аннотации `@Scope` кроме самого `scopeName` доступен параметр – `proxyMode`.

По умолчанию его значение `NO` – прокси не создается. Но если указать `INTERFACES` или `TARGET_CLASS`, то под `@Autowired` будет внедряться не сам объект, а генерированный фреймворком прокси.

И когда проксируемый бин имеет скоуп `prototype`, то объект внутри **прокси будет пересоздаваться при каждом обращении**.

```
@Configuration @EnableScheduling
class AppConfig {
    @Bean @Scope(SCOPE_PROTOTYPE)
    public long currentTime() { return System.currentTimeMillis(); }

    // Гипотетический пример, на самом деле proxyMode не работает для примитивов
    @Bean @Scope(scopeName = SCOPE_PROTOTYPE, proxyMode = ScopedProxyMode.TARGET_CLASS)
    public long actualcurrentTime() { return System.currentTimeMillis(); }
}

@Component
class BrokenTalkingClock {
    @Autowired long currentTime;

    @Scheduled(fixedDelay = 1000) void printTime() {
        System.out.println(currentTime);
    }
}

@Component
class TalkingClock implements ApplicationContextAware {
    ApplicationContext context;
    @Override
    public void setApplicationContext(ApplicationContext context) { this.context = context; }

    @Scheduled(fixedDelay = 1000) void printTime() {
        System.out.println(context.getBean("currentTime"));
    }
}

@Component
class SimpleTalkingClock {
    @Autowired long actualcurrentTime;

    @Scheduled(fixedDelay = 1000) void printTime() {
        System.out.println(actualcurrentTime);
    }
}
```

## Можно ли вставить бин в статическое поле? Почему?

### ПРОБЛЕМА

Spring не позволяет вводить значение в статические переменные, например:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class GlobalValue {

    @Value("${mongodb.db}")
    public static String DATABASE;

}
```

Печатаем  
**GlobalValue.DATABASE**, отображается null

```
GlobalValue.DATABASE = null
```

### РЕШЕНИЕ

Чтобы исправить это, **создайте «не статический сеттер»**, чтобы присвоить введенное значение для статической переменной. Например :

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class GlobalValue {

    public static String DATABASE;

    @Value("${mongodb.db}")
    public void setDatabase(String db) {
        DATABASE = db;
    }
}
```

Output

```
GlobalValue.DATABASE = "mongodb database name"
```

<https://mkyong.com/spring/spring-inject-a-value-into-static-variables/>

### Расскажите про аннотации `@Primary` и `@Qualifier`

`@Primary` используется, чтобы отдавать предпочтение бину, когда есть несколько бинов одного типа. Эта аннотация полезна, когда нужно указать, какой компонент определенного типа должен внедряться по умолчанию.

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 ОБЕ аннотации `@Qualifier` и `@Primary`, то аннотация `@Qualifier` будет иметь приоритет.

По сути, `@Primary` определяет значение по умолчанию, в то время как `@Qualifier` более специфичен: определяет приоритет внедрения.

По умолчанию Spring распознает объекты для вставки по ТИПУ. Если в контейнере доступно **более одного бина одного и того же типа**, будет выброшено исключение.

Для избежания этого можно указать аннотацию Spring Framework `@Qualifier("название поля")`, где название 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.

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

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

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



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.

## Другой пример **@Qualifier**



Field	Setter	Constructor
<code>@Autowired @Qualifier("dog") private Pet pet;</code>	<code>@Autowired @Qualifier("dog") public void setPet(Pet pet) {     this.pet = pet; }</code>	<code>@Autowired public Person(@Qualifier("dog") Pet pet) {     this.pet = pet; }</code>

Если при использовании **@Autowired** подходящих по типу бинов больше одного, то выбрасывается исключение. Предотвратить выброс данного исключения можно, конкретно указав, какой бин должен быть внедрён. **Используют аннотацию **@Qualifier****

## Как зайнжектить примитив?

### **@Value**

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

Понадобится файл свойств (\*.properties), чтобы определить значения, которые нужно внедрить аннотацией **@Value**.

```

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

Сначала в классе конфигурации нужно указать аннотацию **@PropertySource** с именем файла свойств.

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 можно 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;
```

Можно использовать значение поля из другого бина. Предположим, 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;
```

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

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

## @Value with Map

Можно использовать аннотацию @Value для добавления свойств в Map. Для начала нужно определить свойство в формате {key: 'value'} в файле свойств:

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

Теперь можно вставить это значение из файла свойств в виде карты:

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

Можно просто внедрить значение по ключу:

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

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

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

Также можно 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, нужно получить только те записи, значения которых больше единицы:

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

Также можно использовать аннотацию @Value для добавления всех текущих системных свойств:

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

## @Value with Constructor

Можно 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 использовано 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
}

```

## Как заинжектить коллекцию?

Можно внедрять массивы примитивов и ссылочных типов. Со всеми массивами и коллекциями используем внедрение через:

- поля
- конструкторы
- методы-сеттеры

## Array Injection

```

@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 можно 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 необходимо 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.

Например, использован параметр 'name' у аннотации @Bean, чтобы указать конкретный классификатор для бина.

**Элемент 'name' является идентификатором бина**, который должен быть уникальным, потому что все бины хранятся в контейнере в Map.

В случае с коллекцией необходимо, чтобы несколько бинов имели одно и то же имя квалификатора @Qualifier, чтобы их можно было внедрить в одну коллекцию с одним и тем же квалификатором.

В этом случае необходимо 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 СПИСОК (не Set или Map) или массив.

Поддерживаются как аннотация `@Order`, так и **интерфейс Ordered**.

Пример кода →

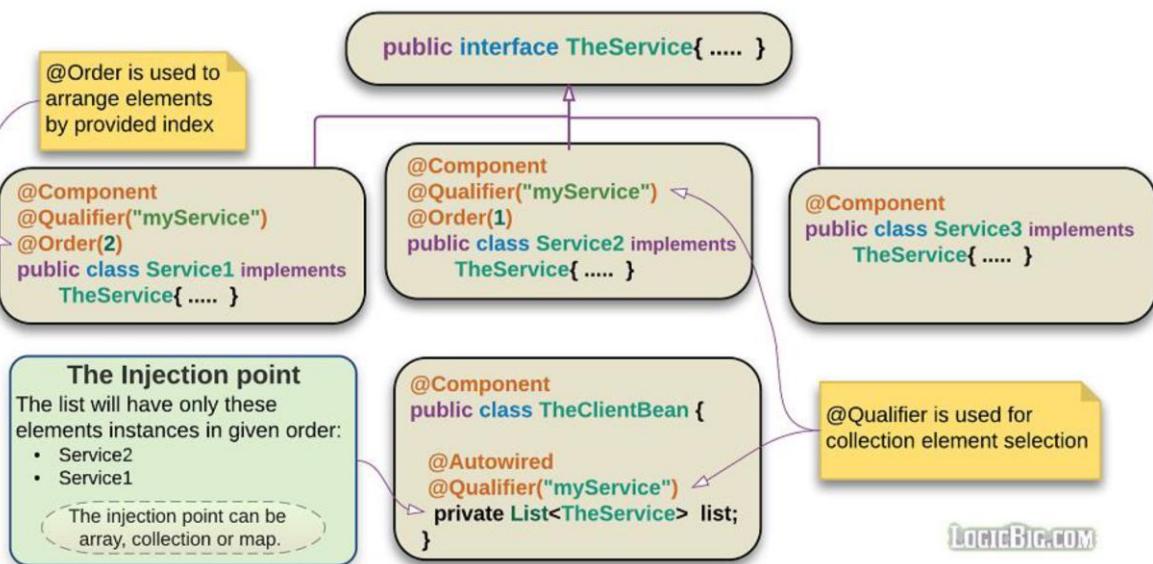
Массив строк будет выглядеть так:

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

```
@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;
        }
    }
}
```

## Injecting Collections/Arrays



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

```

@Service
@Qualifier("myService")
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;
    }
}

```

Расскажите про аннотацию `@Conditional`

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 определяются в специально создаваемых 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, необходимо передать этот класс в аннотацию `@Conditional` в качестве параметра:

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

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

### Расскажите про аннотацию `@Profile`

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

Можно активировать разные профили в разных средах, чтобы загрузить **только те бины, которые нам нужны**. Используя аннотацию `@Profile`, относим бин к конкретному профилю. Эту аннотацию 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. Имеем бин, который должен быть активным только во время разработки, но не должен использоваться в продакшне. Аннотируем этот компонент с профилем «dev», и он будет присутствовать в контейнере только во время разработки, а во время продакшена профиль dev просто НЕ будет активен:

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

В качестве быстрого обозначения имен профилей также могут начинаться с оператора NOT, например «!dev», чтобы исключить их из профиля.

В приведенном 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».

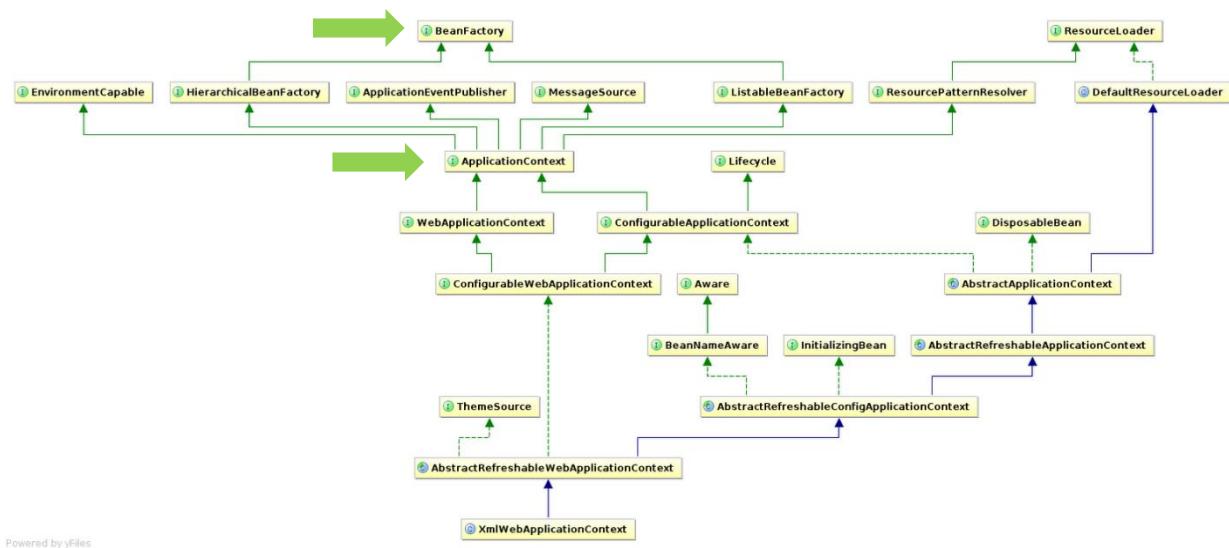
В Spring Boot есть возможность иметь один файл настроек `application.properties`, в котором будут основные настройки для всех профилей, и иметь по файлу настроек для каждого профиля **application-dev.properties** и **application-prod.properties**, содержащие свои собственные дополнительные настройки.

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

`BeanFactory` — это интерфейс, который предоставляет механизм конфигурации, способный управлять объектами любого типа, предоставляет инфраструктуру конфигурации и основные функциональные возможности. Это центральный игрок, отвечает за создание и хранение всех объектов (синглтонов).

`ApplicationContext` расширяет `BeanFactory` и полностью реализует его функционал, добавляя больше специфических функций: предоставляет информацию о конфигурации приложения и услуги транзакций, АОП, является источником сообщений для интернационализации (i18n) и обработки событий в приложении.

**Он доступен только для чтения во время выполнения.**



### ApplicationContext предоставляет:

- Фабричные методы бина для доступа к компонентам приложения
- Возможность загружать файловые ресурсы в общем виде
- Возможность публиковать события и регистрировать обработчики на них
- Возможность работать с сообщениями с поддержкой интернационализации
- Наследование от родительского контекста

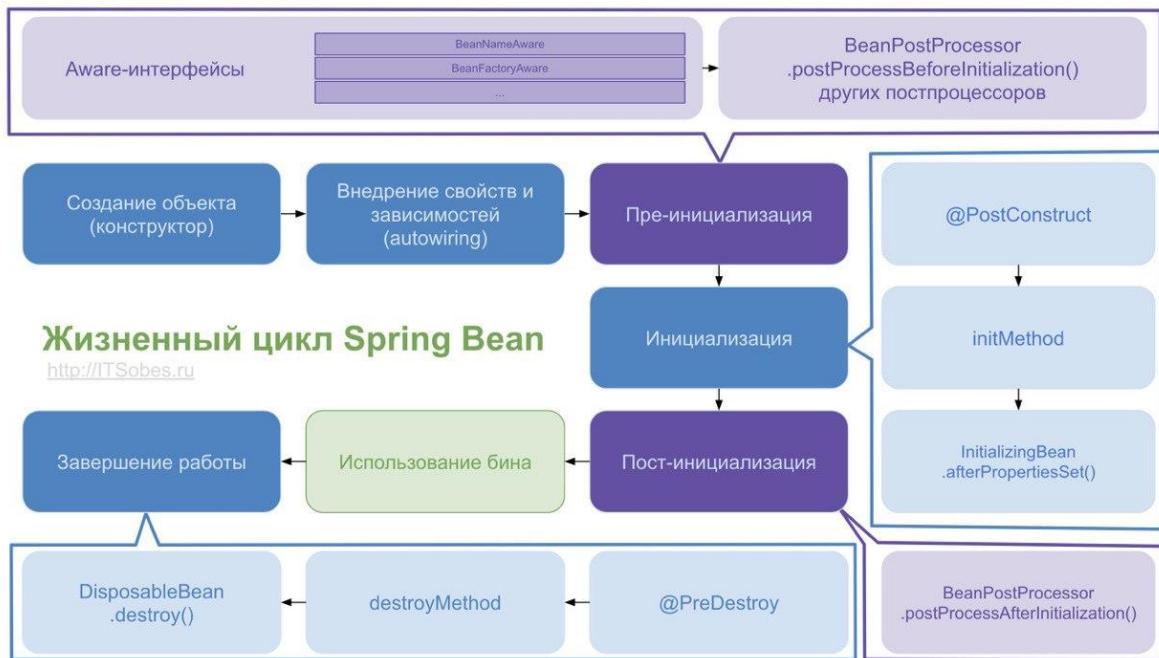
Bean Factory	Application Context
Manual BeanPostProcessor registration	Automatic BeanPostProcessor registration
Support Lazy loading	Support Aggressive loading
Does not support the Annotation based dependency Injection	Support Annotation based dependency Injection.-@Autowired, @PreDestroy
Doesn't Support I18N	Support I18N
it doesn't allow configure to multiple configuration files.  BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));	it allow to configure multiple configuration files.  ApplicationContext context = new ClassPathXmlApplicationContext({ "spring-dao.xml", "spring-service.xml"});
Создание/связывание бина. <u>Ручная регистрация BeanPostProcessor</u>	Создание/связывание бина. <u>Автоматическая регистрация BeanPostProcessor</u> . <u>Автоматическая регистрация BeanFactoryPostProcessor</u> .
<b>Поддержка ленивой загрузки</b> — экземпляр bean-компонента создается при вызове метода <code>getBean()</code> . Это нормально для тестирования и непроизводственного использования. Он существует для обратной совместимости.	<b>Поддержка агрессивной загрузки</b> — он создает экземпляр <b>Singleton bean</b> при запуске контейнера, он НЕ ждет вызова <code>getBean()</code> .  Удобный доступ к <code>MessageSource</code> (для i18n).

	<p>Публикация <b>ApplicationEvent</b> — может публиковать события для bean-компонентов, которые зарегистрированы как слушатели.</p> <p>Поддержка внедрения зависимостей на основе аннотаций. <b>@Autowired, @PreDestroy</b></p> <p>Поддержка многих корпоративных сервисов, таких как доступ JNDI, интеграция EJB, удаленное взаимодействие.</p>
--	--

Расскажите про жизненный цикл бина, аннотации **@PostConstruct** и **@PreDestroy**

### Bean – центральный объект заботы Spring Framework.

За кулисами фреймворка с ними происходит множество процессов. Во многие из них можно вмешаться, добавив собственную логику в разные этапы жизненного цикла.



### Создаём и конфигурируем Bean

```
@Component
public class BeanConfig {

    @Bean(initMethod = "init")
    public NewBean newBean(BeanService beanService) {
        return new NewBean(beanService);
    }
}
```

Пустой класс Service

```
3 usages
@Service
public class BeanService {
    // пустой
}
```

Console

```
initialization completed in 1251 ms
Вход в конструктор
Выход из конструктора
отработал postConstruct метод
отработал init метод
```

### Запускаем Bean

```
2 usages
public class NewBean {
    1 usage
    private final BeanService beanService;

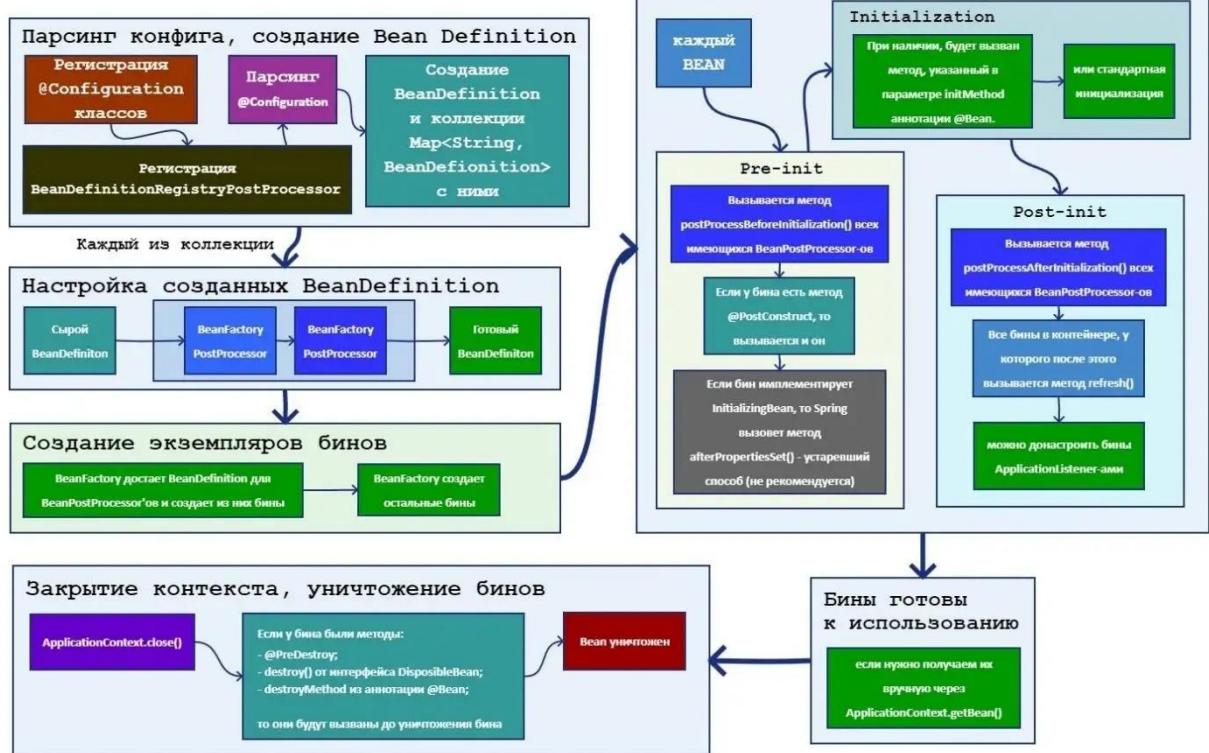
    1 usage
    public NewBean(BeanService beanService) {
        System.out.println("Вход в конструктор");
        this.beanService = beanService;
        System.out.println("Выход из конструктора");
    }

    private void init() {
        System.out.println("отработал init метод");
    }

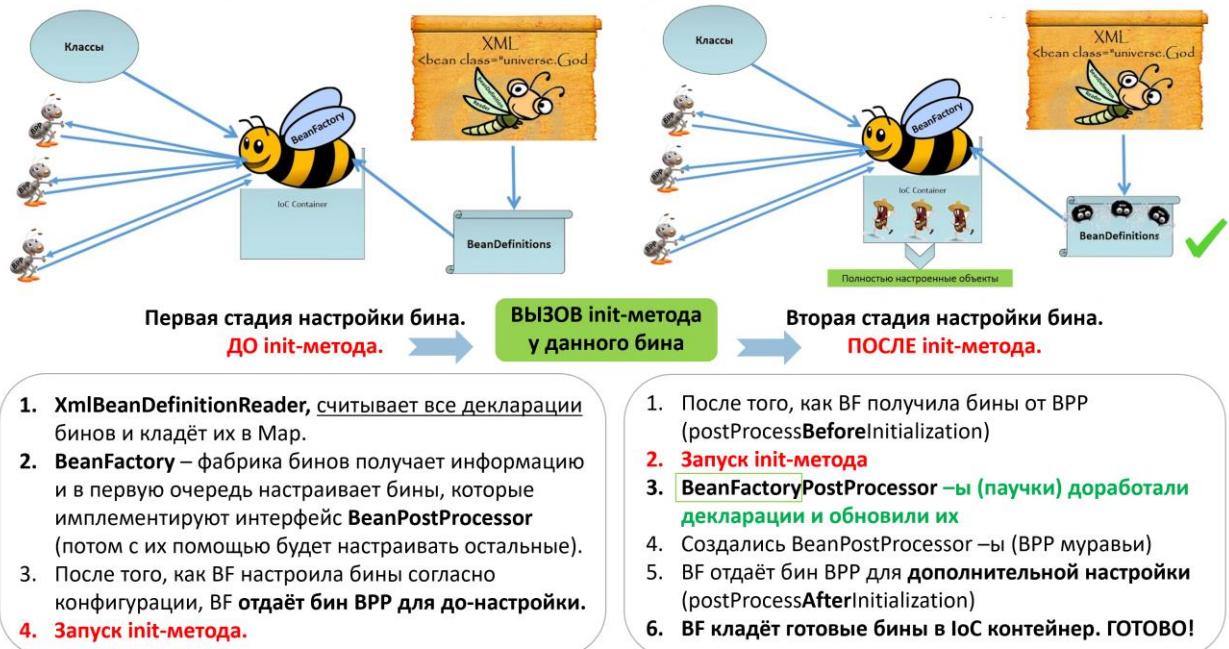
    @PostConstruct
    private void postConstruct() {
        System.out.println("отработал postConstruct метод");
    }
}
```

Для кого-то эта картинка более понятна

## SPRING BEAN жизненный цикл



## Создание и настройка Бина по Евгению Борисову (Спринг потребитель)



Через следующие этапы проходит КАЖДЫЙ отдельно взятый бин:

1. **Инстанцирование объекта** (создание экземпляра класса): техническое начало жизни бина, работа конструктора его класса.

## 2. Установка свойств из конфигурации бина, внедрение зависимостей.

**3. Нотификация aware\*-интерфейсов:** BeanNameAware, BeanFactoryAware и других. Технически, выполняется системными подтипов BeanPostProcessor, и совпадает с п.4;

\*Маркерный интерфейс Aware служит родителем большому количеству интерфейсов с именами \*Aware.

Каждый из них, при реализации, доставляет бину какую-то специфичную для себя сущность. Так, например, компонент, которому нужно обратиться к контексту приложения, должен реализовывать ApplicationContextAware.

Технически, сами интерфейсы ничего не делают. Интерфейс FooAware обычно объявляет единственный метод void setFoo(Foo value). Через этот метод связанный с интерфейсом BeanPostProcessor передаст в бин нужную сущность.

Если бин реализует ServletContextAware, то в процессе инициализации бина к нему придет ServletContextAwareProcessor, и вызовет setServletContext с контекстом сервлета в качестве параметра.

**4. Пре-инициализация** – метод postProcessBeforeInitialization() интерфейса BeanPostProcessor;

**5. Инициализация.** Разные способы применяются в таком порядке:

Метод бина с аннотацией @PostConstruct из стандарта JSR-250 (рекомендуется этот)

Метод afterPropertiesSet() бина под интерфейсом InitializingBean

**Init-метод для отдельного бина** его имя устанавливается в параметре определения initMethod. В xml-конфигурации можно установить для всех бинов сразу, с помощью default-init-method.

**6. Пост-инициализация** – метод postProcessAfterInitialization() интерфейса BeanPostProcessor.

Когда IoC-контейнер завершает свою работу, мы можем кастомизировать этап штатного уничтожения бина. Как со всеми способами финализации в Java, при жестком выключении (kill -9) гарантии вызова этого этапа нет.

Три альтернативных способа «де-инициализации» вызываются в том же порядке, что симметричные им методы инициализации:

- Метод с аннотацией @PreDestroy
- Метод с именем, которое указано в свойстве destroyMethod определения бина (или в глобальном default-destroy-method)
- Метод destroy() интерфейса DisposableBean.

Не следует путать жизненный цикл отдельного бина с жизненным циклом контекста и этапами подготовки фабрик бинов. О них отдельно.

# Методы init и destroy

```
public void init() {
    System.out.println("Class Dog: init method");
}

public void destroy() {
    System.out.println("Class Dog: destroy method");
}

<bean id="myPet"
      class="ioc.Dog"
      init-method="init"
      destroy-method="destroy">
</bean>
```

У данных методов access modifier может быть любым

У данных методов return type может быть любым. Но из-за того, что возвращаемое значение мы никак не можем использовать, чаще всего return type – это void.

Называться данные методы могут как угодно.

В данных методах не должно быть параметров.

Если у бина scope = prototype, то:

- init-method будет вызываться для каждого новосозданного бина.
- для этого бина destroy-method вызываться не будет
- программисту необходимо самостоятельно писать код для закрытия/освобождения ресурсов, которые были использованы в бине

Расскажите про скоупы бинов? Какой скоуп используется по умолчанию?

Документация тут → <https://docs.spring.io/spring-framework/docs/3.0.0.M4/reference/html/ch03s05.html>

Сфера	Описание
<b>singleton</b>	Применяет определение <u>одного компонента к одному</u> экземпляру объекта для каждого контейнера Spring IoC.
<b>prototype</b>	Применяет определение <u>одного компонента к любому количеству</u> экземпляров объекта.
<b>request</b>	Привязывает одно определение компонента <u>к жизненному циклу одного HTTP-запроса</u> ; то есть каждый HTTP-запрос имеет свой собственный экземпляр bean-компонента, созданный на основе одного определения bean-компонента.
<b>session</b>	Привязывает одно определение компонента <u>к жизненному циклу HTTP Session</u> .
<b>global session</b>	Охватывает одно определение bean-компонента <u>с жизненным циклом глобального HTTP Session</u> . Обычно действует только при использовании в контексте портлета.

Request, session, global session действительны только в контексте WEB-ориентированного Spring ApplicationContext.

Scope (область видимости) определяет:

- жизненный цикл бина
- возможное количество создаваемых бинов

Разновидности bean scope:

singleton

prototype

request

session

global-session

В Spring Framework во всех определениях бизнес-сущностей (bean) явно или неявно присутствует атрибут scope. В Java-конфигурации он передается в аннотации `@Scope`, в xml – в атрибуте scope тега `<bean>`.

Атрибут scope – это строка-идентификатор, которая ставит бину в соответствие экземпляр класса `org.springframework.beans.factory.config.Scope`.

**Скоуп** – реализация паттерна «стратегия» для фабрик бинов, инструкция по созданию бизнес-объектов.

В простейшем Spring-приложении всегда существует два скоупа:

- **singleton** – объект создается однажды, при последующих внедрениях переиспользуется. Полезен для большинства случаев: различные сервисы, объекты без состояния, неизменяемые объекты.

Это не класс-синглтон: при объявлении двух бинов одного класса их экземпляров будет два. **Это скоуп по умолчанию**.

- **prototype** – при каждом внедрении фабрика бинов создает новый объект.  
Нужен для изменяемых бинов с состоянием.

Spring Web добавляет 4 дополнительных скоупа, которые делают бин синглтоном в пределах обработки одного сетевого запроса (*request*), клиентской сессии (*session*), контекста сервлета (*application*) и вебсокет-сессии (*websocket*).

### Разработчик может добавлять собственные скоупы.

Пример реализации одного можно найти в самих исходниках Spring:  
`SimpleThreadScope`, который делает бин тред-локальным.

Для использования его, как и пользовательские скоупы, нужно сначала зарегистрировать в `BeanFactory`.

## Bean scope

`singletone` – дефолтный scope.

- такой бин создаётся сразу после прочтения Spring Container-ом конфиг файла.
- является общим для всех, кто запросит его у Spring Container-а.
- подходит для stateless объектов.

`prototype`

- такой бин создаётся только после обращения к Spring Container-у с помощью метода `getBean`.
- для каждого такого обращения создаётся новый бин в Spring Container-е.
- подходит для stateful объектов.

```
@Component
```

```
@Scope ("singletone")
```

```
public class Dog implements Pet {
```

```
@Component
```

```
@Scope ("prototype")
```

```
public class Dog implements Pet {
```

### SINGLETONE

`singletone` – дефолтный scope.

- такой бин **создаётся сразу после прочтения Spring Container-ом конфиг файла**.
- является общим для всех, кто запросит его у Spring Container-а.
- подходит для stateless объектов.

```
<bean id="myPet"
      class="ioc.Dog"
      scope="singleton">
</bean>
```

Spring Container

Dog `myDog` = context.getBean(name: "myPet", Dog.class);  
Dog `yourDog` = context.getBean(name: "myPet", Dog.class);

### PROTOTYPE

`prototype`

- такой бин **создаётся только после обращения к Spring Container-у с помощью метода `getBean`**.
- для каждого такого обращения **создаётся новый бин в Spring Container-е**.
- подходит для stateful объектов.

```
<bean id="myPet"
      class="ioc.Dog"
      scope="prototype">
</bean>
```

Spring Container

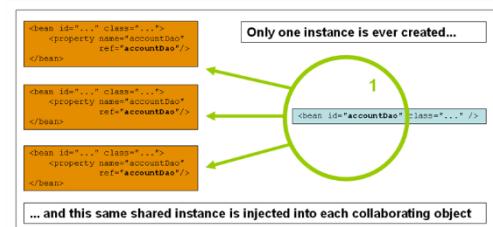
Dog `myDog` = context.getBean(name: "myPet", Dog.class);  
Dog `yourDog` = context.getBean(name: "myPet", Dog.class);

## СИНГЛТОН «под капотом»

Управляется только один общий экземпляр одноэлементного компонента, и все запросы компонентов с идентификатором или идентификаторами, совпадающими с определением этого компонента, приводят к тому, что контейнер Spring возвращает **этот конкретный экземпляр компонента**.

Иными словами, когда вы определяете определение bean-компонента и оно **ограничено областью действия синглтона**, контейнер Spring IoC создает ровно один экземпляр объекта, определенного этим определением bean-компонента.

Этот единственный экземпляр **хранится в кэше** таких одноэлементных компонентов, и все последующие запросы и ссылки для этого именованного компонента **возвращают кэшированный объект**.

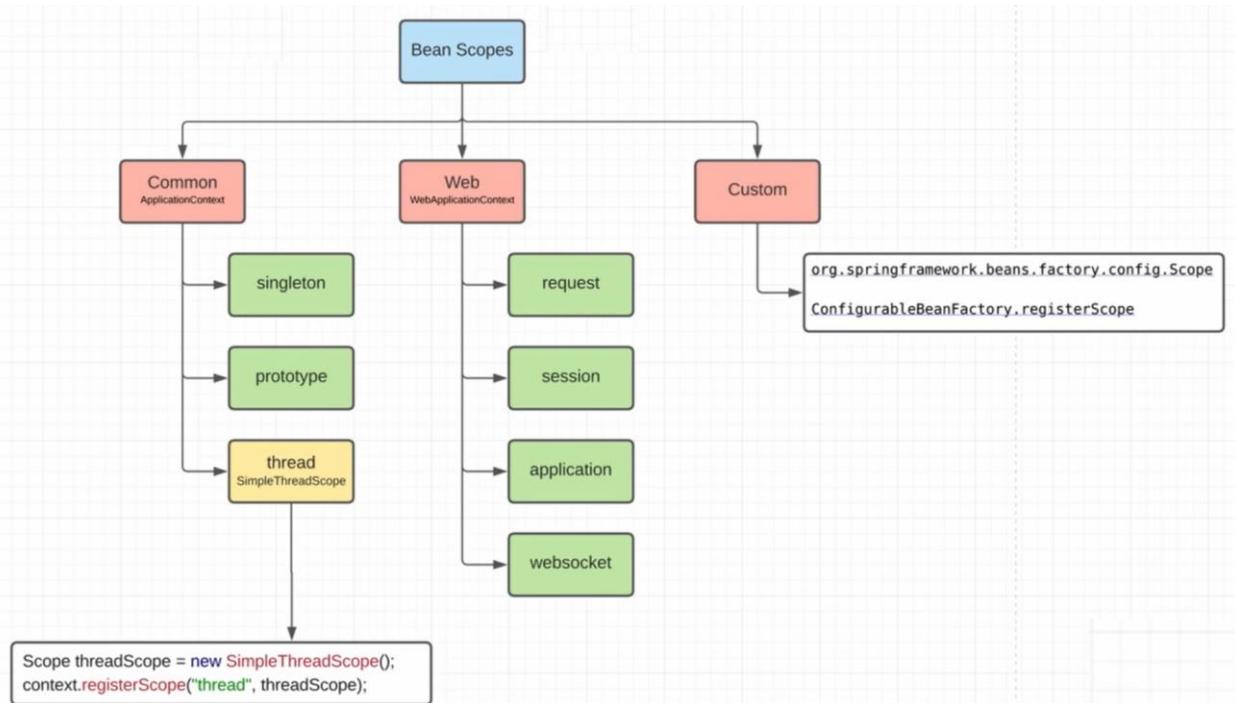
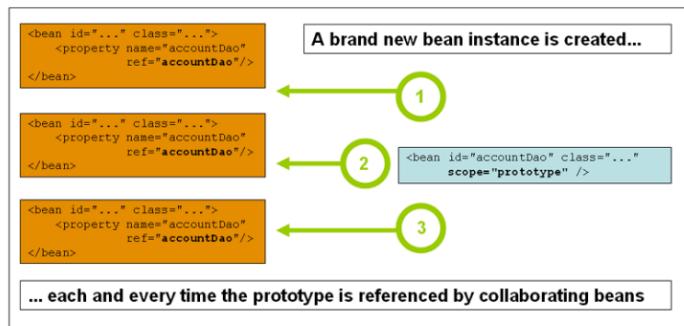


## ПРОТОТАЙП «под капотом»

Не одноэлементная, prototype scope развертывания компонента приводит к **созданию нового экземпляра компонента каждый раз, когда делается запрос на этот конкретный компонент**.

То есть компонент вводится в другой компонент или вы запрашиваете его через вызов метода `getBean()` в контейнере. Как правило, используйте область прототипа для всех bean-компонентов с состоянием и область singleton для bean-компонентов без сохранения состояния.

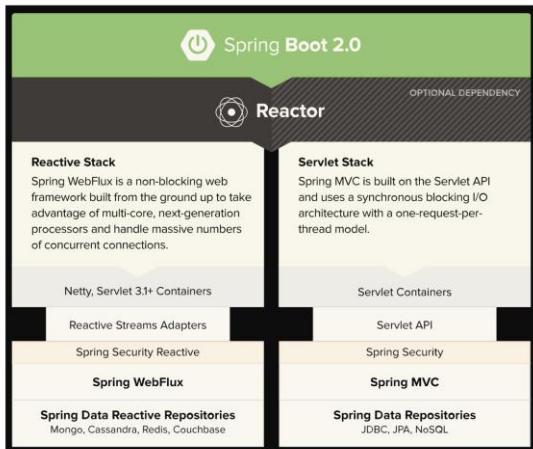
Следующая диаграмма иллюстрирует область действия прототипа Spring. Объект доступа к данным (DAO) обычно не настраивается в качестве прототипа, поскольку типичный DAO не поддерживает никакого диалогового состояния; просто автору было проще повторно использовать ядро одноэлементной диаграммы.



## Что изменилось в пятом спринге?

<https://www.youtube.com/watch?v=1I03BB2gteU>

- Используется JDK 8+ (Optional, CompletableFuture, Time API, java.util.function, default methods)
- Поддержка Java 9 (Automatic-Module-Name in 5.0, module-info in 5.1+, ASM 6)
- Поддержка HTTP/2, NIO/NIO.2, Kotlin
- Spring-Data-JPA 2.x, Spring-Security 5.x
- Прочие изменения
  - Null-safety аннотации (@Nullable), новая документация
  - Component index at compilation time (alternative to classpath scanning)
  - Совместимость с Java EE 8 (Servlet 4.0, Bean Validation 2.0, JPA 2.2, JSON Binding API 1.0)
  - Новый common logging bridge - spring-jcl
  - Поддержка JUnit 5 + Testing Improvements (conditional and concurrent)
  - Удалена поддержка: Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava
- Реактивность
  - Flux, Mono, Reactor 3, Spring WebFlux
  - Адаптеры для Reactor, RxJava 1, RxJava 2, CompletableFuture и Java 9+ Flow.Publisher



## Реактивность

- Spring 5 uses Reactor 3
- Reactor 3 project
- Разница между RxJava и API Java 9 Flow API
- Web on Reactive Stack
- [Пример на Spring WebFlux](#)
- New in Spring 5: Functional Web Framework

[Расскажите про аннотацию @ComponentScan](#)

Благодаря аннотации `@ComponentScan` происходит сканирование бин-компонентов.



Если необходимо, чтобы Spring использовал только определенный набор классов компонентов, то нужно использовать аннотацию `@ComponentScan`, которая будет указывать **на конкретное расположение классов bean-компонентов**, которые Spring должен будет проинициализировать.

Эту аннотацию можно использовать с параметрами или без них.

Без параметров Spring будет сканировать текущий пакет и его подпакеты, а при параметризации сообщит Spring, где именно искать пакеты.

```
@Configuration
@ComponentScan(
    basePackages = {"com.baeldung демопакет"},
    basePackageClasses = DemoBean.class)
public class ComponentScanExample {
    // ...
}
```

По умолчанию классы, аннотированные `@Component`, `@Repository`, `@Service`, `@Controller` регистрируются как Spring bean -компоненты. То же самое касается классов, аннотированных пользовательской аннотацией с помощью `@Component`

Есть возможность расширить это поведение, используя параметры `includeFilters` и `excludeFilters` аннотации `@ComponentScan`

Для `ComponentScan.Filter` доступно пять типов фильтров:

ANNOTATION	Тип фильтра АННОТАЦИЯ <u>включает</u> или <u>исключает</u> классы в сканах компонентов, которые отмечены заданными аннотациями.
ASSIGNABLE_TYPE	Фильтрует все классы во время сканирования компонентов, которые либо расширяют класс, либо реализуют интерфейс указанного типа.
REGEX	Фильтр проверяет, <u>соответствует ли имя класса заданному шаблону</u> регулярного выражения. FilterType.REGEX проверяет как простые, так и полные имена классов.
ASPECTJ	Когда мы хотим использовать выражения для <u>выбора сложного подмножества классов</u> , нам нужно использовать ASPECTJ FilterType
CUSTOM	Если ни один из приведенных выше типов фильтров не соответствует нашим требованиям, мы также можем <u>создать собственный тип фильтра</u> . Например, предположим, что мы хотим сканировать только те классы, имя которых не превышает пяти символов.

Примеры кода и статья по ссылке → <https://www.baeldung.com/spring-componentscan-filter-type>

```
@Configuration
@ComponentScan(includeFilters = @ComponentScan.Filter(type = FilterType.ANNOTATION,
    classes = Animal.class))
public class ComponentScanAnnotationFilterApp { }
```

Как мы видим, сканер отлично распознает нашего Слона:

Давайте используем `FilterType.ASPECTJ`, чтобы указать Spring для сканирования классов, соответствующие регулярному выражению `.Intl`. Наше регулярное выражение оценивает все, что содержит `nt`:

```
@Configuration
@ComponentScan(includeFilters = @ComponentScan.Filter(type = FilterType.REGEX,
    pattern = ".*(nt)"))
public class ComponentScanRegexFilterApp { }
```

На этот раз в нашем тесте мы увидим, что Spring сканирует `Elephant`, но не `Lion`:

Давайте используем `FilterType.CUSTOM`, чтобы передать Spring для сканирования классов с использованием нашего пользовательского фильтра `ComponentScanCustomFilter`:

```
@Configuration
@ComponentScan(includeFilters = @ComponentScan.Filter(type = FilterType.CUSTOM,
    classes = ComponentScanCustomFilter.class))
public class ComponentScanCustomFilterApp { }
```

Хотя это немного сложно, наша логика здесь требует, чтобы bean-компоненты не начинались ни с «Л», ни с «С» в своем имени класса, так что это снова оставляет нас с `Elephant`:

## Как спринг работает с транзакциями?

\*Здесь не рассматриваются распределенные транзакции или реактивные транзакции, хотя общие принципы применимы.

Вспомним, что такие транзакции и принципы ACID т.к. Spring использует принципы JDBC

**Транзакция – это единица работы в рамках соединения с базой данных.**

Для запуска необходимо подключение к базе данных.

Имеет ДВА состояния:

- Выполняется полностью – **commit**
- Откатывается полностью – **rollback**



Уровень изолированности транзакций — условное значение, определяющее, в какой мере в результате выполнения логически параллельных транзакций в СУБД допускается получение несогласованных данных.

Уровень изоляции	Фантомное чтение	Неповторяющееся чтение	«Грязное» чтение	Потерянное обновление <sup>[3]</sup>
SERIALIZABLE	✓	✓	✓	✓
REPEATABLE READ	✗	✓	✓	✓
READ COMMITTED	✗	✗	✓	✓
READ UNCOMMITTED	✗	✗	✗	✓

Вот так Spring устанавливает уровни изоляции для соединения с базой данных:

```
import java.sql.Connection;  
  
// isolation=TransactionDefinition.ISOLATION_READ_UNCOMMITTED  
connection.setTransactionIsolation(Connection.TRANSACTION_READ_UNCOMMITTED); // (1)  
  
// propagation=TransactionDefinition.NESTED  
  
...  
  
connection.rollback(savePoint);
```

Вложенные транзакции в Spring — это просто **точки сохранения JDBC** / базы данных.  
Поддержка **Savepoints** (точек сохранения) зависит от драйвера JDBC.

Метод `Connection.setSavepoint` устанавливает объект `Savepoint` в текущей транзакции.  
Метод `Connection.rollback` перегружен, чтобы принимать аргумент точки сохранения.

В то время как в обычном JDBC есть только один способ управлять транзакциями,  
`setAutocommit(false)`, Spring предлагает множество различных, более удобных  
способов добиться того же самого.

**ПРОГРАММНЫЙ** способ определения транзакций в Spring (редко используется)

- либо через транзакционный шаблон **TransactionTemplate**
- либо непосредственно через **PlatformTransactionManager**

По сравнению с обычным примером JDBC:

- Вам не нужно возиться с открытием и закрытием соединений с базой данных самостоятельно (`try-finally`). Вместо этого вы используете обратные вызовы транзакций.
- Вам также не нужно ловить `SQLExceptions`, поскольку **Spring** преобразует эти исключения в исключения времени выполнения (`runtime exceptions`).
- Кроме того, вы лучше интегрируетесь в экосистему **Spring**. `TransactionTemplate` будет использовать `TransactionManager` внутри, который будет использовать источник данных. Все это бины, которые вы должны указать в конфигурации контекста **Spring**, но о которых вам больше не придётся беспокоиться в дальнейшем.

```
@Service  
public class UserService {  
  
    @Autowired  
    private TransactionTemplate template;  
  
    public Long registerUser(User user) {  
        Long id = template.execute(status -> {  
            // выполнить некоторый SQL, который, например,  
            // вставляет пользователя в базу данных  
            // и возвращает автогенерированный идентификатор  
            return id;  
        });  
    }  
}
```

## ДЕКЛАРАТИВНЫЙ

Управление транзакциями с помощью XML – это устаревший способ, поскольку он был вытеснен гораздо более простой аннотацией `@Transactional` (современный способ).

[Расскажите про аннотацию `@Transactional`](#)

Мы можем использовать `@Transactional` для включения метода в транзакцию БД.

Это позволяет нам установить условия распространения, изоляции, тайм-аута, только для чтения и отката для транзакции. Мы также можем указать менеджера транзакций.

### Детали реализации:

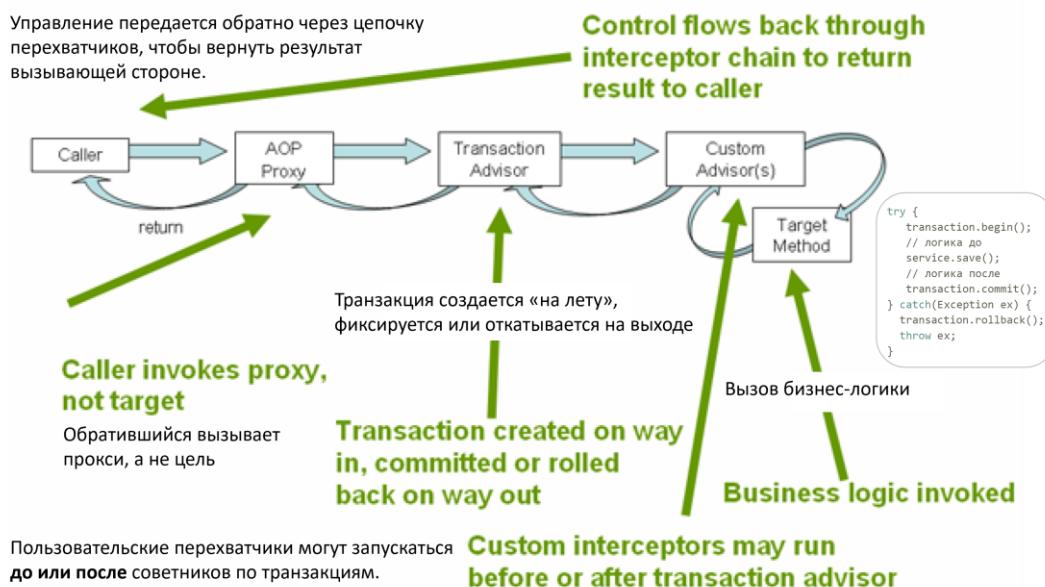
Spring создает прокси или манипулирует байт-кодом класса для управления созданием, фиксацией и откатом транзакции.

В случае прокси Spring игнорирует `@Transactional` во внутренних вызовах методов. Если у нас есть такой метод, как `callMethod`, и мы помечаем его как `@Transactional`, Spring обернет некоторый код управления транзакциями вокруг вызываемого метода `@Transactional` →

```
createTransactionIfNecessary();  
try {  
    callMethod();  
    commitTransactionAfterReturning();  
} catch (exception) {  
    completeTransactionAfterThrowing();  
    throw exception;  
}
```

## Декларативное управление транзакциями, Declarative Transaction Management.

Для работы с транзакциями Spring Framework использует AOP-прокси:



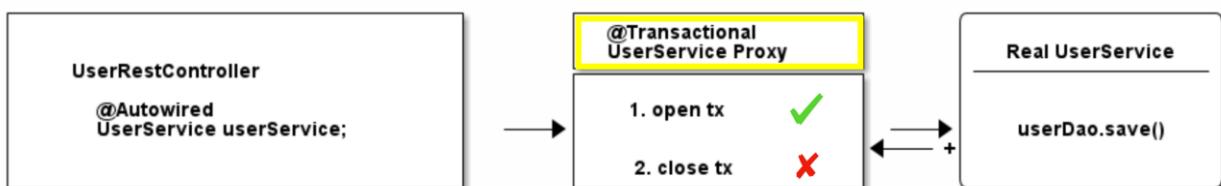
Для включения возможности управления транзакциями первым делом нужно разместить аннотацию `@EnableTransactionManagement` у класса-конфигурации `@Configuration`

Это означает, что **классы, помеченные `@Transactional`, должны быть обернуты АСПЕКТОМ транзакций**. Однако, если 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.



UserService проксируется на лету, и **прокси управляет транзакциями** для вас. Но НЕ сам прокси управляет всем этим транзакционным состоянием (открыть, зафиксировать, закрыть), прокси **делегирует эту работу менеджеру транзакций**.

Spring предлагает интерфейс **PlatformTransactionManager / TransactionManager**, который по умолчанию поставляется с парой удобных реализаций. Одна из них – это **менеджер транзакций источника данных**.

Менеджер транзакций источника данных использует «под капотом» при управлении транзакциями точно такой же код, который мы использовали в разделе **JDBC**.



**ИТАК,**

Если Spring обнаруживает **@Transactional** на бины, он создаёт динамический PROXY на эти бины (обёртку).

PROXY имеет доступ к **менеджеру транзакций** и будет просить его открывать и закрывать транзакции/соединения.

Сам менеджер транзакций будет просто делать то, что обычно: управлять соединением JDBC.

- **Физические транзакции:** это фактические транзакции JDBC.
- **Логические транзакции:** это (потенциально вложенные) аннотированные **@Transactional** методы Spring.

<https://habr.com/ru/post/682362/>

Spring теперь должен быть достаточно умным, чтобы две логические транзакции (`invoice() / createPdf()`) отображались на две разные физические транзакции базы данных:

Изменение режима распространения (propagation) на `requires_new` говорит Spring, что `createPDF()` должна выполняться в собственной транзакции, независимо от любой другой, уже существующей транзакции.

Это означает, что ваш код будет открывать **два** (физических) соединения/транзакции с базой данных. (Снова: `getConnection() x2 . setAutocommit(false) x2 . commit() x2`.)

```

@Service
public class InvoiceService {
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void createPDF() {
        // ...
    }
}
  
```

## What are **@Transactional Propagation Levels** used for?

Изучая код Spring, вы найдете множество уровней или режимов распространения, которые можно подключить к методу **@Transactional**.

```

@Transactional(propagation = Propagation.REQUIRED)
// или

@Transactional(propagation = Propagation.REQUIRES_NEW)
// и т.д.
  
```

<https://www.baeldung.com/spring-transactional-propagation-isolation>

<b>Required (по умолчанию)</b> необходимый	Моему методу нужна транзакция, либо откройте ее для меня, либо используйте существующую → <code>getConnection().setAutocommit(false).commit()</code>  Spring проверяет, есть ли активная транзакция, и, если ничего не существует, создает новую. В противном случае бизнес-логика добавляется к текущей активной транзакции.
<b>Supports</b> поддерживаемый	Мне не важно, открыта транзакция или нет, я могу работать в любом случае → не имеет отношения к JDBC  SUPPORTS Spring сначала проверяет, существует ли активная транзакция. Если транзакция существует, то будет использована существующая транзакция. Если транзакции нет, она выполняется не-транзакционно.
<b>Mandatory</b> обязательный	Я не собираюсь открывать транзакцию сам, но я буду плакать, если никто другой не откроет её → не имеет отношения к JDBC  При распространении MANDATORY, если есть активная транзакция, она будет использована. Если активной транзакции нет, Spring выдает исключение: <code>throw IllegalStateException;</code>

<b>Requires_new</b>	Я хочу полностью собственную транзакцию → getConnection().setAutocommit(false).commit()  REQUIRES_NEW, Spring приостанавливает текущую транзакцию, если она существует, а затем создает новую.
<b>Not_Supported</b> не поддерживаемый	Мне очень не нравятся транзакции, я даже попытаюсь приостановить текущую, запущенную транзакцию → ничего общего с JDBC  Если текущая транзакция существует, Spring сначала приостанавливает ее, а затем выполняется бизнес-логика без транзакции.
<b>Never</b> никогда	Я буду плакать, если кто-то другой запустит транзакцию → не имеет отношения к JDBC Spring выдает исключение, если есть активная транзакция.
<b>Nested</b> вложенный	Это звучит так сложно, но мы просто говорим о точках сохранения! → connection.setSavepoint()  Spring проверяет, существует ли транзакция, и если да, то отмечает точку сохранения. Это означает, что, если выполнение нашей бизнес-логики выдает исключение, транзакция откатывается к этой точке сохранения save point. Если активной транзакции нет, она работает как REQUIRED.

Большинство режимов, уровней распространения Propagation Levels не имеют ничего общего с базой данных или JDBC, но больше с тем, **как вы структурируете свою программу со Spring: КАК, ГДЕ и КОГДА Spring ожидает присутствия транзакций.**

В этом случае Spring будет ожидать, что транзакция будет открыта, когда вы вызовете myMethod() класса UserService.

Он не открывает её сам, вместо этого, если вызвать этот метод без предварительно существующей транзакции, Spring выбросит исключение. Это дополнительные нюансы при "логической обработке транзакций".

```
public class UserService {
    @Transactional(propagation = Propagation.MANDATORY)
    public void myMethod() {
        // выполнить sql
    }
}
```

## Подводные камни

Есть класс UserService с транзакционным методом invoice(), который вызывает createPDF(), который также является транзакционным.

**Сколько физических транзакций будет открыто, когда кто-то вызовет invoice()?** Нет, ответ будет не два, а один. Почему?

Spring создает транзакционный прокси UserService, но как только мы оказываемся внутри класса UserService и вызываем другие внутренние методы, **прокси больше не задействован**. Это означает, что новой транзакции не будет.

```
@Service
public class UserService {

    @Transactional
    public void invoice() {
        createPdf();
        // отправка счета по электронной почте и т.д.
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void createPdf() {
        // ...
    }
}
```

Есть несколько приёмов (например, самоинъекция), которые можно использовать, чтобы обойти это ограничение. **Но главный вывод: всегда помните о ГРАНИЦАХ транзакций PROXY.**

[Расскажите про аннотации @Controller и @RestController. Чем они отличаются?](#)

**Controller** – это один из стереотипов Spring Framework. Компоненты такого типа обычно занимаются **обработкой сетевых запросов**. Контроллер состоит из набора методов-обработчиков, помеченных аннотацией `@RequestMapping`.

Ответ на запрос можно сформировать разными способами: например, просто вернуть из обработчика строку с именем jsp-файла, или же вернуть `ResponseBodyEmitter`, который будет асинхронно заполняться данными позже. Все возможные варианты перечислены в документации. <https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-return-types>

Большинство современных API реализуется по архитектуре REST. В ней каждая сущность доступна под собственным URI\*.

**URI — унифицированный идентификатор ресурса**, последовательность символов, идентифицирующая абстрактный или физический ресурс. Ранее назывался Universal Resource Identifier — универсальный идентификатор ресурса.

В методе-обработчике возвращается экземпляр класса этой сущности, который преобразуется в ответ сервера одним из `HttpMessageConverter`-ов. Например, в JSON его превратит `MappingJackson2HttpMessageConverter`.

Чтобы использовать этот способ ответа, метод, или весь контроллер, должен иметь аннотацию `@ResponseBody`.

**@RestController** – просто сокращенная запись для `@Controller + @ResponseBody`.

**@Controller** – это обычная аннотация, которая используется для пометки класса как Spring MVC Controller а **@RestController** – это специальный контроллер, используемый в веб-сервисах RESTful, и эквивалент `@Controller + @ResponseBody`.

Аннотация `@ResponseBody` ставится на методы, которые работают с данными, а не с моделями. Ее не требуется указывать явно, если используется `@RestController`.

Обычные методы возвращают Model, а методы аннотированные `@ResponseBody` возвращают объекты, которые конвертируются в медиафайлы с помощью `HttpMessageConverter`.

### Как вернуть ответ со своим статусом (например, 213)?

Можно создавать свои ошибки, указывая через аннотации код ответа сервера и сообщение об ошибке.

В случае выкидывания кодом такой ошибки в процессе обработки запроса спринг автоматически преобразует ее в JSON ответ со всей нужной информацией и код ответа будет соответствующий.

Если же надо изменить **код для ответа без ошибки**, то можно так (код на java):

Шаг 1: указываем возвращаемый тип метода контроллера как  `ResponseEntity<?>`

Шаг 2: возвращаем из метода контроллера

```
new ResponseEntity<>(someObjectThatWillBeSerializedToJsonBySpring, HttpStatus.CREATED);
```

ИЛИ  `ResponseEntity.status(201).body(someObjectThatWillBeSerializedToJsonBySpring)`

Либо можно попробовать добавить аннотацию для метода контроллера:  
`@ResponseStatus(HttpStatus.CREATED)`

### Что такое ViewResolver?

Все платформы MVC предоставляют **способ работы с представлениями View**.

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 некоторыми другими.

По умолчанию реализацией интерфейса ViewResolver является класс InternalResourceViewResolver. Любым реализациям ViewResolver желательно поддерживать интернационализацию, то есть множество языков.

[Чем отличаются Model, ModelMap и ModelAndView?](#)

## Model

ИНТЕРФЕЙС, лежит в пакете **spring-context**. В методах контроллера можно 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**.

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

Имеет все преимущества LinkedHashMap плюс несколько удобных методов:

## ModelAndView

Этот КЛАСС лежит в пакете **spring-webmvc** и может одновременно хранить модели и представление, чтобы контроллер мог отдавать их в ОДНОМ возвращаемом значении.

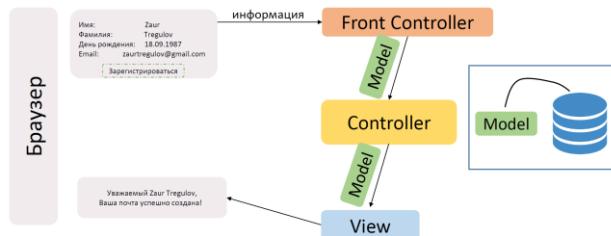
Внутри содержит поле `private Object view`, куда записывает нужное представление, а также поле `private ModelMap model`, куда и складывает все атрибуты модели →

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

[Расскажите про паттерн MVC, как он реализован в Spring?](#)



## Пример Spring MVC



View – web страница, которую можно создать с помощью html, jsp, Thymeleaf и т.д.

Часто при отображении View использует данные из Model.

JSP – Java Server Page. Представляет собой html код с небольшим добавлением кода Java

JSTL - Java Server Pages Standard Tag Library. Это расширение спецификации JSP.

## Spring MVC

Front Controller также известен под именем DispatcherServlet. Он является частью Spring.

Controller – центр управления, мозг Spring MVC приложения.

Model – контейнер для хранения данных.

Из чего будет состоять Spring MVC приложение:

- Конфигурация Spring
- Описание Spring бинов
- Web страницы

Расскажите про паттерн Front Controller, как он реализован в Spring?

Паттерн Front Controller обеспечивает единую точку входа для всех входящих запросов.

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.**

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.

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”.

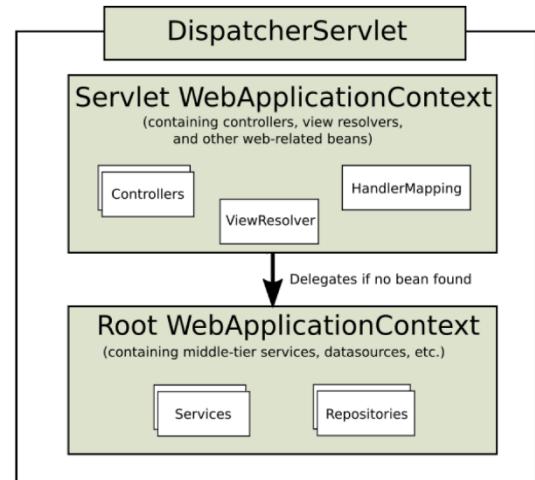
Spring предоставляет нам такую возможность, и для начала нужно понять, что:

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** 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 необходимо 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**.

Package [org.springframework.web.context](#)

Class **ContextLoaderListener**

`java.lang.Object`  
`org.springframework.web.context.ContextLoader`  
`org.springframework.web.context.ContextLoaderListener`

All Implemented Interfaces:

`ServletContextListener` , `EventListener`

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"`

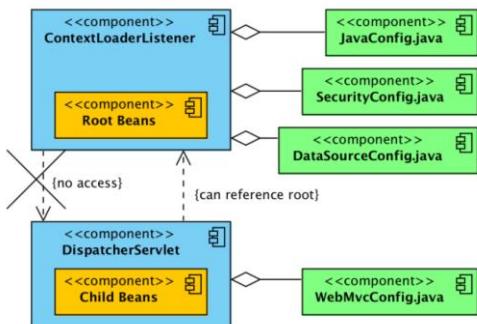
Чтобы из контроллера любого дочернего контекста обратиться к корневому контексту приложения, можно использовать класс **WebApplicationContextUtils**, содержащий статические методы →

```
@Autowired
ServletContext context;

ApplicationContext ac =
    WebApplicationContextUtils.getWebApplicationContext(context);

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

**ContextLoaderListener** создает корневой контекст приложения.



1. Каждый DispatcherServlet создаёт себе один дочерний контекст.
2. Дочерние контексты могут обращаться к бинам, определенным в корневом контексте.
3. Бины в корневом контексте НЕ могут получить доступ к бинам в дочерних контекстах (напрямую).
4. Все контексты добавляются в ServletContext.
5. Получить доступ к корневому контексту можно, используя класс **WebApplicationContextUtils**

## Что такое AOP? Как реализовано в спринге?

Аспектно-ориентированное программирование AOP — 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**.

AOP предоставляет возможность реализации в одном месте **сквозной логики**. То есть логики, которая применяется к множеству частей приложения и обеспечение автоматического применения этой логики по всему приложению.

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

Статья → <https://habr.com/ru/post/428548/>

## PROXY что это, для чего и как применяется в Spring

**Паттерн Заместитель (Proxy)** предоставляет объект-заместитель, который управляет доступом к другому объекту. То есть создается объект-суррогат, который может выступать в роли другого объекта и замещать его (перехватывать все вызовы).

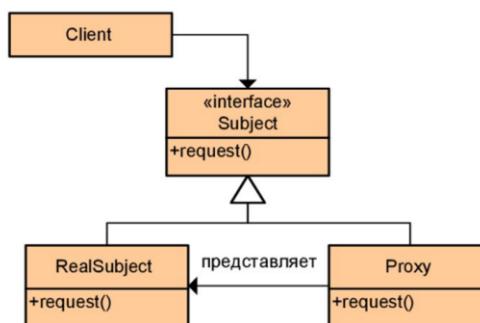
## Справка: паттерн проектирования PROXY

### Прокси Proxy

**Тип:** Структурный

**Что это:**

Представляет замену другого объекта для контроля доступа к нему.



**Заместитель PROXY — структурный шаблон проектирования.**

Представляет объект-суррогат, который контролирует доступ к другому объекту, перехватывая все вызовы.

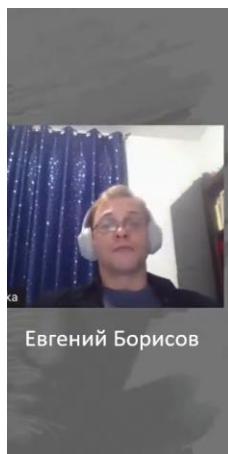
В Spring прокси просто обращает bean, он может добавить логику до и после выполнения методов.

Spring AOP использует либо динамические прокси JDK (на основе интерфейса), либо CGLIB (на основе класса) для создания прокси для данного целевого объекта.

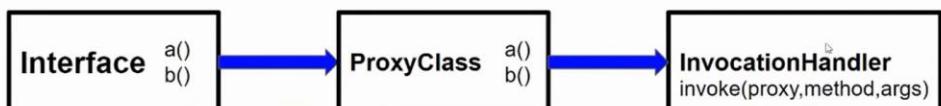
**Динамические прокси-серверы JDK** предпочтительнее, когда у вас есть выбор.

Если целевой объект для проксирования реализует хотя бы один **интерфейс**, то будет использоваться динамический прокси JDK. Все интерфейсы, реализованные целевым типом, будут проксированы.

<https://youtu.be/DKNDU7OjyJs?t=541>



### Как работает dynamic proxy?

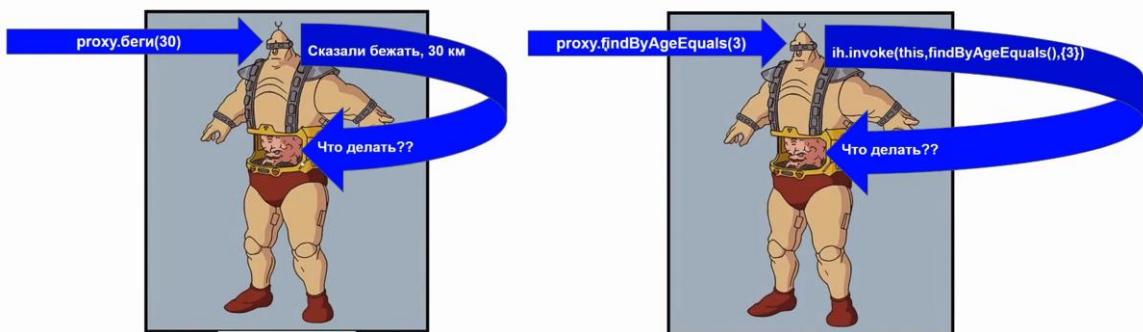


1. Пишем интерфейс

2. Создаём **Proxy Class** с помощью Java библиотеки, который будет имплементировать те же самые методы

3. Реализовывать эти методы будет InvocationHandler, который мы при создании Proxy передаём ему. Тут будет один метод, кот. принимает сигнатуру метода, кот. вызвали у прокси класса, аргументы, которые ему передали и сам объект прокси.

Для всех PROXY – это умный, классный объект, который офигенно работает, а по факту это **ТУПОЙ объект**, который ничего не умеет сам, а за него думает **InvocationHandler**.



Если целевой объект **не реализует** интерфейсов, будет создан прокси-сервер CGLIB.

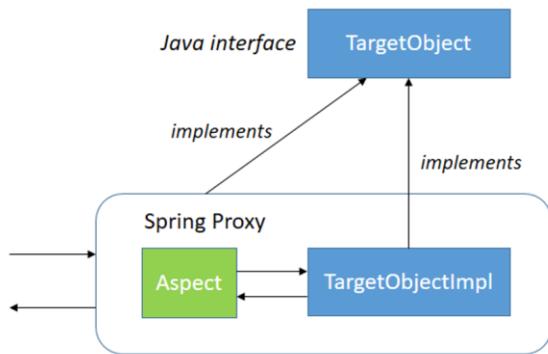
Если вы хотите принудительно использовать проксирование CGLIB (например, проксировать каждый метод, определенный для целевого объекта, а не только те, которые реализованы его интерфейсами), вы можете это сделать.

Чтобы принудительно использовать прокси CGLIB, установите для атрибута `proxy-target-class` элемента `<aop:config>` значение `true`.

## Spring AOP Process

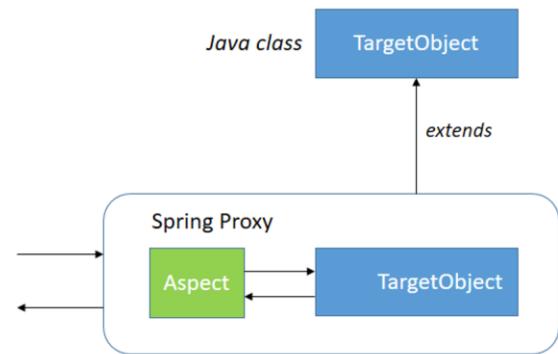
### JDK Proxy (interface based)

JDK dynamic proxy - Spring AOP по умолчанию использует JDK dynamic proxy, которые позволяют проксировать любой интерфейс (или набор интерфейсов). Если целевой объект реализует хотя бы один интерфейс, то будет использоваться динамический прокси JDK.



### CGLib Proxy (class based)

CGLIB-прокси - используется по умолчанию, если бизнес-объект НЕ реализует ни одного интерфейса.



## В чем разница между Filters, Listeners and Interceptors?

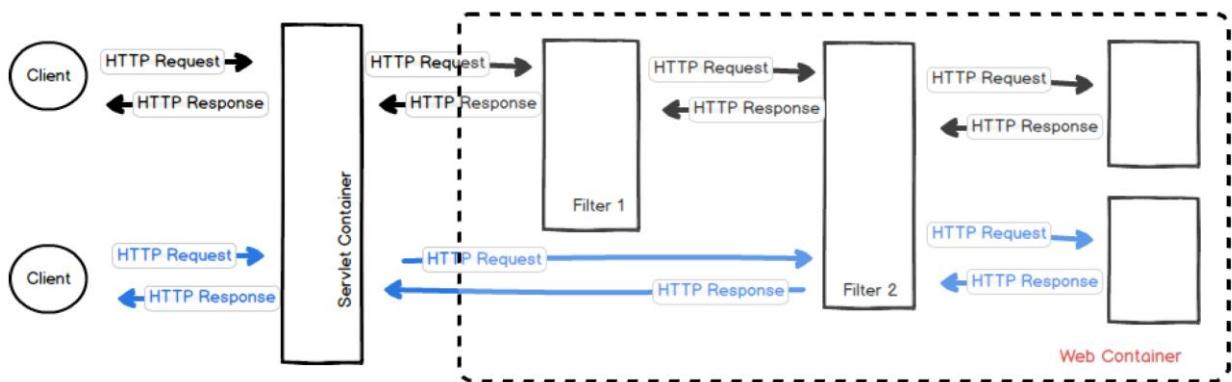
### Filter

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.

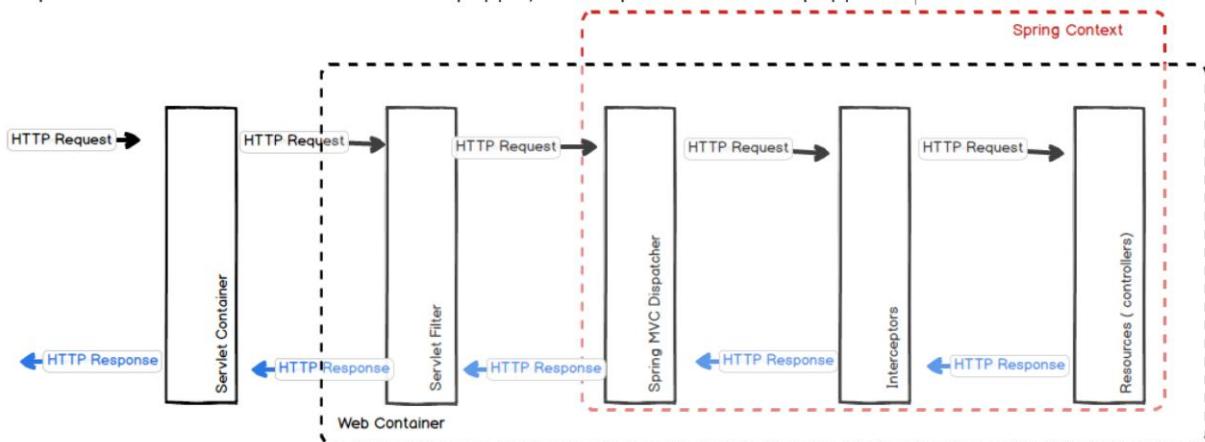


### Interceptor

Это ИНТЕРФЕЙС из пакета **org.aopalliance.intercept**, предназначенный для АОП 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. Можно использовать Interceptor для выполнения таких задач, как запись в Log, добавление или обновление конфигурации перед тем, как запрос обрабатывается Controllerом.

**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.



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

Перехватчики работают с **HandlerMapping** и поэтому должны реализовывать интерфейс **HandlerInterceptor** или наследоваться от готового класса **HandlerInterceptorAdapter**.

В случае реализации **HandlerInterceptor** нужно переопределить 3 метода, а в случае **HandlerInterceptor**, только те, что необходимы:

- `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 исполнение дальше по цепочке перехватчиков до метода контроллера.

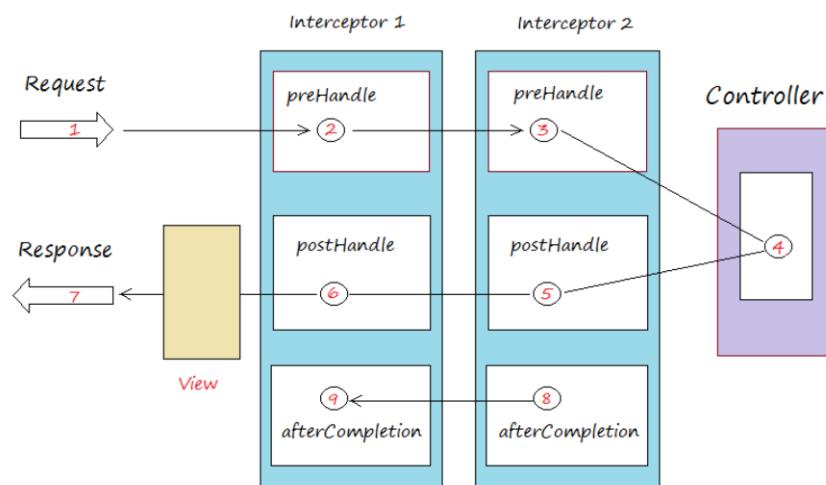
Если этот метод возвращает `true`, то запрос отправляется следующему перехватчику или в контроллер. Если метод возвращает `false`, то исполнение запроса прекращается, обычно отправляя ошибку HTTP или записывая собственный ответ в `response`.

- `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`!



`HandlerInterceptor` связан с бином `DefaultAnnotationHandlerMapping`, который отвечает за применение перехватчиков к любому классу, помеченному аннотацией `@Controller`.

Чтобы добавить перехватчики в конфигурацию Spring, 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:

- Фильтры до
- Перехватчики до
- **Метод контроллера**
- Перехватчики после
- Фильтры после

**HandlerInterceptor** в основном похож на Servlet Filter, но в отличие от него 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 **HandlerInterceptor -s являются бинами** и могут обращаться к другим компонентам в приложении.

**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, необходимо инициализировать пул соединений с базой данных до запуска веб-приложения. **ServletContextListener** – это то, что нужно. Он будет запускать код до запуска веб-приложения.

Все **ServletContextListener -s** уведомляются об инициализации контекста до инициализации любых фильтров или сервлетов в веб-приложении, а также уведомляются об уничтожении контекста после того, как все сервлеты и фильтры уничтожены.

Чтобы создать свой Listener, 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");
    }
}
```

Можно ли передать в запросе один и тот же параметр несколько раз? Как?

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

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

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

IDs are [1,2,3]

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

Зачем нужен такой модуль?

- Настройка безопасности
- Защита отдельных методов или целых блоков приложения

Конфигурация с помощью аннотации

`@EnableWebSecurity (default @Configuration)`

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    private final SuccessUserHandler successUserHandler;
    private final UserServiceImpl userService;
```

Под капотом цепочка – реализуем цепочку фильтров (в коде это стрим):

Для аутентификации используем POST запрос  
и обязательные поля сущности:

- username
- password

Spring ждёт эти поля и будет генерировать  
если явно их не обозначим. Когда эти данные  
«летят» в `HttpServletRequest`, то фильтры  
будут выдёргивать именно эти данные.

Необходимо использовать **csrf токен** – это  
защита от скрытых полей и редиректа на  
вредоносные, чужие базы.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .antMatchers("/", "/index").permitAll()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers("/user").hasAnyRole("USER", "ADMIN")
        .anyRequest().authenticated()
        .and()
        .formLogin().successHandler(successUserHandler)
        .permitAll()
        .and()
        .logout()
        .permitAll();}
```

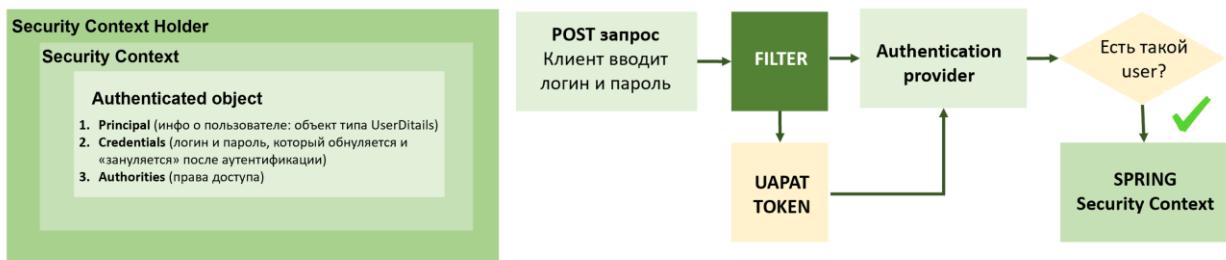
**Principal principal** – это информация о текущем пользователе, которая хранится в контексте Spring Security:  
**principal.getName()** – так получаем имя аутентифицированного пользователя.  
**Идемпотентность в контроллер-методе.**

### Authenticated object

1. Principal (инфо о пользователе: объект типа UserDitails)
2. Credentials (логин и пароль, который обнуляется и «зануляется» после аутентификации)
3. Authorities (права доступа)

## Авторизация

Где и как хранится инфо о пользователях?



**Идентификация** – уточнение имени и пароля.

**Аутентификация** – это процедура проверки подлинности пользователя путём сравнения введённого им логина и пароля с сохранёнными в базе данных.

**Авторизация** – это проверка и определение полномочий на выполнение определённых действий в соответствии с ранее выполненной аутентификацией.

### Как это работает?

- 1) Пользователь формирует POST запрос
- 2) Фильтр перехватывает и выдёргивает из запроса логин и пароль, далее формирует UAPAT токен (Username And Password Authorization token)
- 3) Токен летит дальше и попадает в Authentication provider. Его задача провести аутентификацию и сказать, существует ли такой пользователь?  
И если да, то кладём его в SPRING Security Context.

**Зачем тут нужны UserService и Authentication provider?**

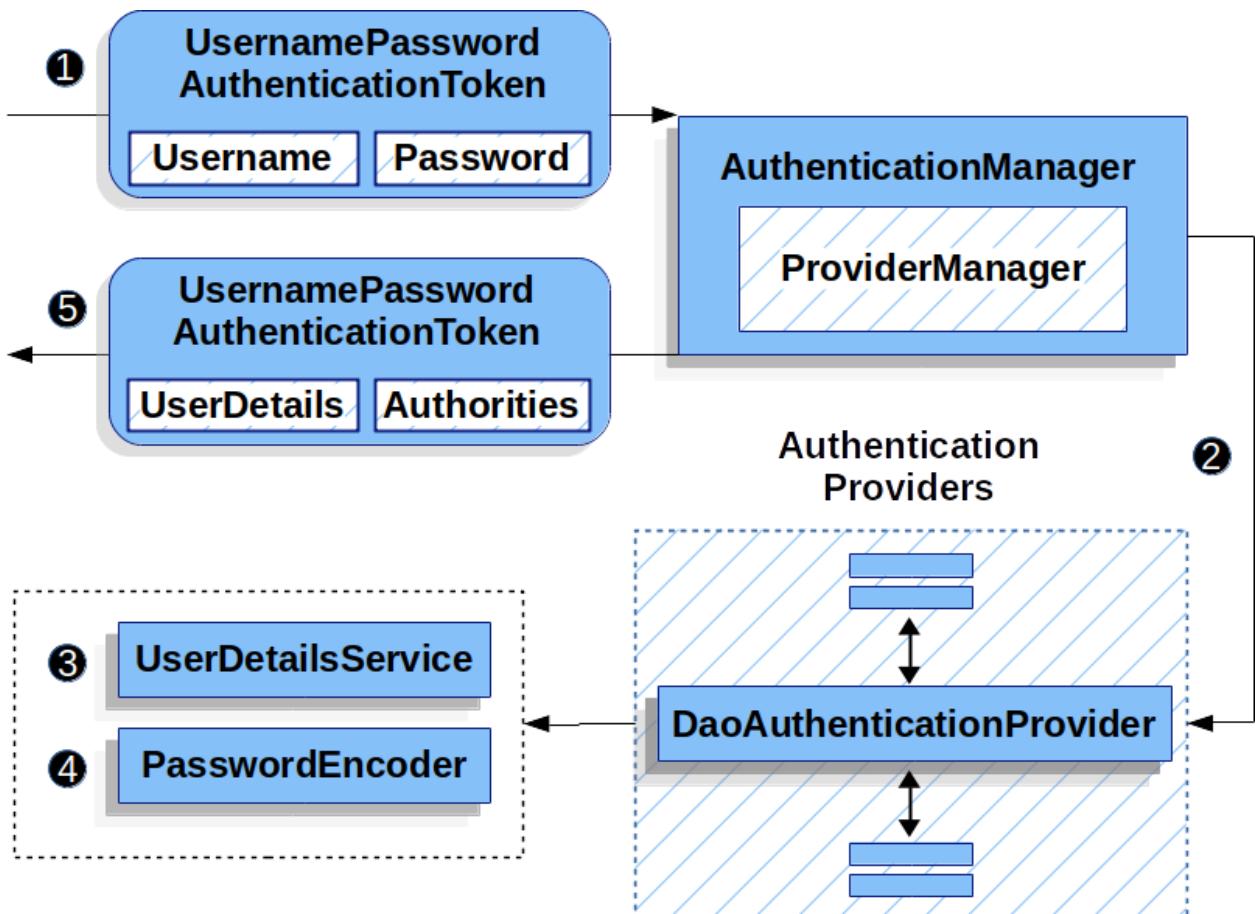
- Провайдер получает токен (логин и пароль).
- Провайдер обращается к **UserDitailsService**: есть у тебя такой юзер?
- UserDitailsService знает, где юзеры хранятся и по username (byUserName) получает и возвращает настоящего юзера в виде объекта UserDitails (principal).
- Далее Authentication provider берёт из токена пароль и делает проверку **equals()**
- Если они равны, кладёт их в Spring Security Context.

Задачи Authentication provider: запросить настоящего юзера у UserDitailsService, сравнить пароль, **почистить credentials** (логин и пароль не сохраняются).

Spring хранит только **Authorities** (права доступа), а если нужна полная информация о клиенте, то нужно обратиться в базу данных.

### ФИКСИРУЕМ

**DaoAuthenticationProvider** — это реализация **AuthenticationProvider**, которая использует **UserDetailsService** и **PasswordEncoder** для проверки подлинности имени пользователя и пароля.



Когда пользователь посылает POST request, то «формируется посылка». КАК?

№1 В цепочке фильтров стоит фильтр, который обрабатывает и выдёргивает TOKEN

№2 Далее действует **Authentication Manager**, в котором есть **Provider Manager**. Он может состоять не из одного провайдера, а из многих, настроенных на разные реакции.

В **DaoAuthenticationProvider** записаны:

- №3 **UserDetailsService** (возвращает пользователя по имени) и ...
- №4 **PasswordEncoder** (преобразовывает введённый пароль к хэшу)

Если все ОК, то возвращается **UPAT TOKEN**

- **UserDetails** (принципал)
- **Authorities** (права доступа)

Если аутентификация прошла успешно, возвращаемая проверка подлинности имеет тип **UsernamePasswordAuthenticationToken** и имеет принципала, который представляет собой **UserDetails**, возвращаемый настроенной службой **UserDetailsService**.

В конечном итоге возвращенный **UPAT** устанавливается в **SecurityContextHolder** фильтром аутентификации.

Источник: <https://docs.spring.io/spring-security/reference/servlet/authentication/passwords/dao-authentication-provider.html>

## Что такое SpringBoot? Какие у него преимущества? Как конфигурируется? Подробно.

Основные сущности фреймворка Spring Boot – это *стартеры*. Зависимости с названиями вида `spring-boot-starter-xxx` выполняют две основных задачи.

Во-первых, они добавляют набор типичных сторонних библиотек-зависимостей, а во-вторых, регистрируют типичные бины и их конфигурации. Кроме того, со Spring Boot в проекте появляется ряд таких полезностей, как embedded-сервер, конфигурация web-приложения без `web.xml`, метрики, properties вынесенные из кода во внешние файлы.

Например, `spring-boot-starter-data-jpa` даст вам готовый комплект всего необходимого для использования JPA: драйвер, совместимую с ним версию Hibernate, библиотеки Persistence API и Spring Data. В контексте приложения появятся все нужные для JPA репозиториев бины.

Таким образом, Spring Boot ускоряет и упрощает разработку, дает возможность избавиться от boilerplate-кода в проекте и сфокусироваться на бизнес-задачах.

Это бывает особенно важно в микросервисной архитектуре, когда создается большое количество приложений.

С другой стороны, такая избыточность естественно приводит к большей тяжеловесности и **медлительности приложения**.

### Какие задачи решает Spring Data?

Это проект, который упрощает работу с системами доступа к данным: реляционными и нереляционными базами данных, map-reduce фреймворками и облачными хранилищами.

Центральная концепция проекта – *репозитории* из предметно-ориентированного дизайна (Domain-driven design, DDD).

Spring Data состоит из множества отдельных библиотек для разных случаев жизни. Вот самые популярные из них:

- **Spring Data JPA** – адаптер для реализаций Java Persistence API, таких как Hibernate.
- **Spring Data JDBC** – более простой и ограниченный чем JPA адаптер для JDBC-драйверов.
- **Spring Data REST** – создание готовых hypermedia-driven RESTful сервисов на основе репозиториев.
- **Spring Data KeyValue** – работа с хранилищами типа ключ-значение.
- Библиотеки поддержки конкретных реализаций хранилищ: MongoDB, Redis, Cassandra, LDAP, и других.

Основная часть работы в Spring Data строится вокруг интерфейса `Repository`.

Это маркерный интерфейс. От него наследуются интерфейсы-специализации, которые уже содержат методы для работы с сущностями базы данных. Все эти интерфейсы параметризуются двумя типами: самой сущности и её идентификатора.

## Какова иерархия интерфейсов/классов репозитория в Spring Data JPA?

**Repository** (маркерный интерфейс, не содержит методов) — это интерфейс верхнего уровня, определенный в Spring Data Hierarchy.

Интерфейс **CrudRepository** расширяет интерфейс репозитория, предоставляет методы для выполнения операции CRUD.

**PagingAndSortingRepository** расширяет интерфейс CrudRepository и предоставляет дополнительные методы для извлечения сущностей с помощью разбиения на страницы и сортировки.

Интерфейс **QueryByExampleExecutor**, используется для выполнения запроса по примеру.

Интерфейс **JpaRepository** расширяет интерфейсы PagingAndSortingRepository и QueryByExampleExecutor, предоставляя некоторые дополнительные пакетные методы.

**SimpleJpaRepository** — это класс реализации интерфейса CrudRepository.  
**QueryDslJpaRepository** — это класс.

\*[Как преодолеть проблему блокирующих вызовов?](#) (реактивное программирование, WebFlux, EventLoop)

**Требования сегодняшнего дня:** приложения должны иметь высокую доступность и обеспечивать низкое время отклика даже при высокой нагрузке.

### Проблематика

<https://habr.com/ru/post/565004/>

- Модель «поток на запрос»

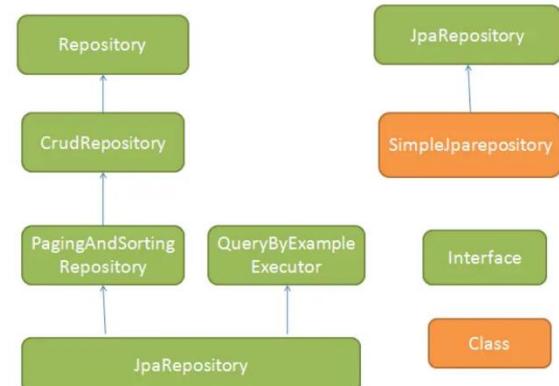
Традиционный способ разработки веб-приложения с помощью Spring — это использование MVC и его развертывание в контейнере сервлетов, таком как Tomcat. Контейнер сервлетов имеет выделенный пул потоков для обработки HTTP-запросов, где каждому входящему запросу будет назначен поток, и этот поток будет обрабатывать весь жизненный цикл запроса (модель «поток на запрос»). По умолчанию для сервера Tomcat установлено 200 подключений.

Это означает, что приложение сможет обрабатывать **количество одновременных запросов, равное размеру пула потоков**. Можно настроить размер пула потоков, но поскольку каждый поток резервирует некоторую память (обычно 1 МБ), чем больший размер пула потоков мы настраиваем, тем выше потребление памяти.

Если приложение разработано в соответствии с архитектурой на основе микросервисов, то у нас есть лучшие возможности для масштабирования в зависимости от нагрузки, но **за высокое использование памяти по-прежнему приходится платить** (потоки часто блокируются в ожидании ответа от другой службы, что приводит к огромной трате ресурсов).

- Ожидание операций ввода/вывода

Такой же тип потерь также возникает при ожидании завершения других типов операций ввода-вывода, таких как вызов базы данных или чтение из файла. Во всех этих ситуациях поток, выполняющий запрос ввода-вывода, будет заблокирован и будет ожидать, пока операция ввода-вывода не будет завершена, это называется **блокирующим вводом-выводом**. Такие ситуации, когда выполняющийся поток блокируется, просто ожидая ответа, означают потерю потоков и, следовательно, потерю памяти (рисунок слева).



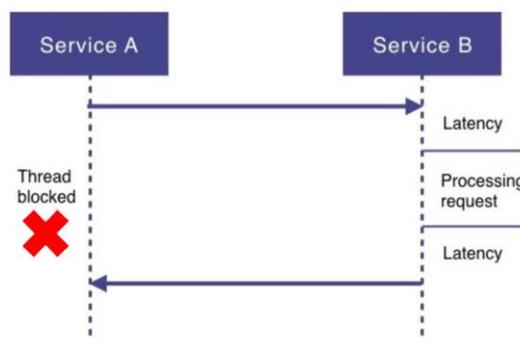


Рисунок 1 - Поток заблокирован в ожидании ответа

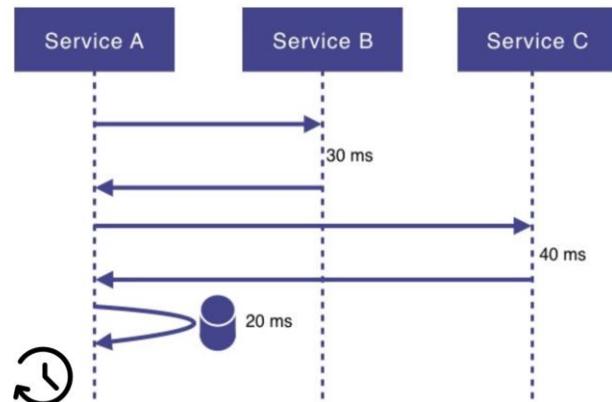


Рисунок 2 - Последовательные вызовы

- Время ответа

Другой проблемой традиционного императивного программирования является **время отклика**, когда службе необходимо выполнить более одного запроса ввода-вывода (рисунок справа). Например, службе А может потребоваться вызвать службы В и С, а также выполнить **поиск в базе данных**, а затем вернуть в результате некоторые агрегированные данные. Это будет означать, что время ответа службы А, помимо времени ее обработки, будет суммой следующих значений:

- время отклика услуги В (задержка сети + обработка)
- время отклика службы С (задержка сети + обработка)
- время ответа на запрос к базе данных (сетевая задержка + обработка)

Если нет никакой реальной логической причины выполнять эти **вызовы последовательно**, то, безусловно, если эти вызовы будут выполняться параллельно, это очень положительно повлияет на время отклика службы А.

- Перегрузка клиента

Другой тип проблемы, которая может возникнуть в ландшафте микросервисов, - когда сервис А запрашивает некоторую информацию у сервиса В, скажем, например, обо всех заказах, размещенных в течение последнего месяца. Если количество заказов окажется огромным, для службы А может возникнуть проблема получить всю эту информацию сразу. Сервис А может быть перегружена большим объемом данных, что может привести, например, к ошибке нехватки памяти.

**Все эти проблемы решает реактивное программирование.**

Перечислим преимущества:

- отходим от модели «поток на запрос» и можем обрабатывать больше запросов с небольшим количеством потоков
- предотвращаем блокировку потоков при ожидании завершения операций ввода-вывода
- упрощаем параллельные вызовы
- поддерживаем «обратное давление», давая клиенту возможность сообщить серверу, с какой нагрузкой он может справиться

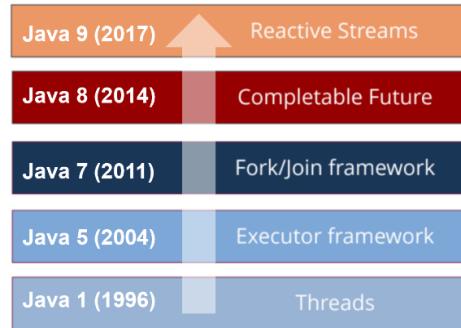
**Реактивное программирование – это асинхронность, соединенная с потоковой обработкой данных**

То есть это не от слова «реактивный = высокоскоростной», это НЕ про ускорение. Это про **оптимизацию управления тредами, потоками**: количество обрабатываемых запросов за единицу времени (от слова «реакция» ... на события).

**Не ждём, а работаем:** реактивность добавляет возможность обрабатывать данные потоком, разбивая задачу на множество подзадач.

Говорят, когда Генри Форд придумал свой конвейер, он повысил производительность труда в четыре раза, благодаря чему ему удалось сделать автомобили доступными.

Здесь мы видим то же самое: у нас не большие порции данных, а конвейер с потоком данных, и каждый обработчик пропускает через себя эти данные, каким-то образом их преобразовывая.



## Что такое реактивный поток?

<https://habr.com/ru/company/oleg-bunin/blog/545702/>

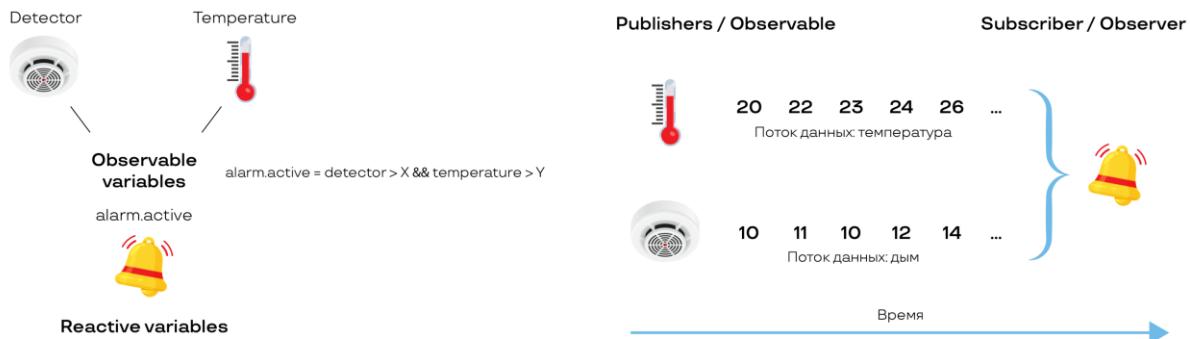
Поток представляет собой некую последовательность, состоящую из постоянных событий, отсортированных по времени. В потоке может быть лишь ТРИ типа сообщений:

- сами данные
- ошибки
- сигнал о завершении работы потока

Для работы **наблюдатель должен лишь подписатьсь на поток**. Это необходимое условие — иначе работа не начнет выполняться.

Один из жизненных примеров реактивности — система оповещения при пожаре.

Используется паттерн **Observer (Наблюдатель)**. Детектор дыма и термометр — это публикаторы сообщений, то есть источники данных (Publisher), а колокольчик на них подписан, то есть он Subscriber, или наблюдатель.



Если бы у нас был традиционный, а не реактивный подход, мы бы писали код, который каждые пять минут опрашивал детектор дыма и датчик температуры, и включает или выключает колокольчик.

Однако в **реактивном подходе** за нас это делает **реактивный фреймворк**, а мы только прописываем условия: колокольчик активен, когда детектор больше X, а температура больше Y. Это происходит каждый раз, когда приходит НОВОЕ СОБЫТИЕ.

Сами данные будут получены лишь тогда, когда они будут готовы — это **event модель**. То есть это может произойти либо в этом же, либо в совершенно другом потоке.

Можно с уверенностью заявить, что **реактивная модель основана на событиях** — это и есть самая настоящая **event-driven модель**.

Получить данные из потока можно двумя способами: **PUSH** и **PULL** модели. **PULL** чтобы получить данные из потока мы сделали запрос на получение данных. Стандартный подход.

PUSH чтобы получить данные, мы не делаем запрос на получение данных. Данные сами нас уведомляют о своей готовности, и поток просто отдает их нам. В это время мы можем не ждать их вообще. Мы просто подписались на поток и забыли про него. Данные готовы к выдаче, отлично.

**Реактивное приложение** — это то, которое само извещает об изменении своего состояния, где мы не делаем запрос и не проверяем, что там изменилось, а приложение само сигнализирует об этом. Это совершенно другой подход в разработке.

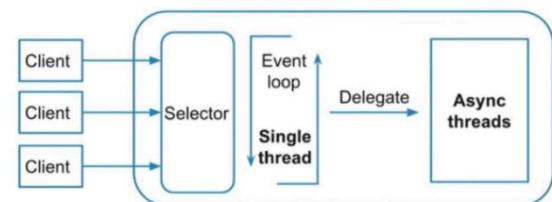
### Netty as a non-blocking server

Рассмотрим пример использования реактивных потоков Flux вместе со Spring Reactor.

В основе Reactor лежит **сервер Netty**. Spring Reactor — это **основа технологии**, которую мы будем использовать. А сама **технология** называется WebFlux.

Чтобы WebFlux (веб флАкс) работал, нужен **асинхронный неблокирующий сервер**.

Есть **Selector** — входной поток, который принимает запросы от клиентов и отправляет их на выполнение в освободившиеся потоки. Если в качестве синхронного сервера (Servlet-контейнера) используется Tomcat, то в качестве **асинхронного** используется Netty.



Драйвер r2dbc от команды Spring сразу предоставляет **реактивное API** (а не асинхронное), а для транспорта **используется Netty** (как и в jasync-sql).

### Blocking vs Reactive

У нас есть два стека обработки запросов:

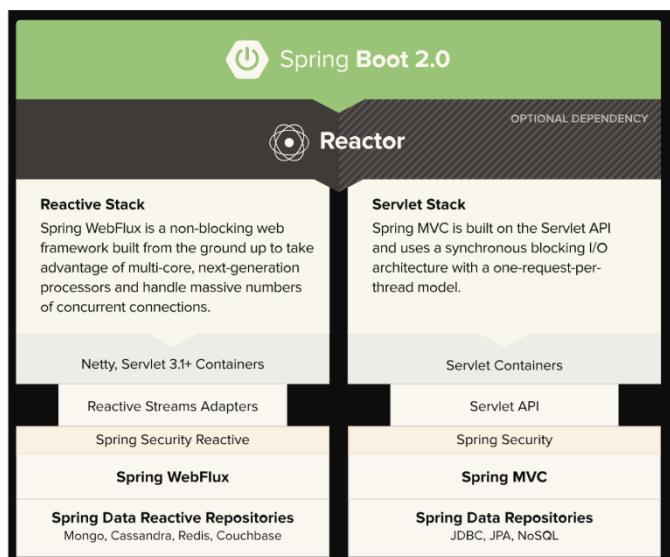
- Традиционный блокирующий стек.
- Неблокирующий стек — в нем все происходит асинхронно и реактивно.

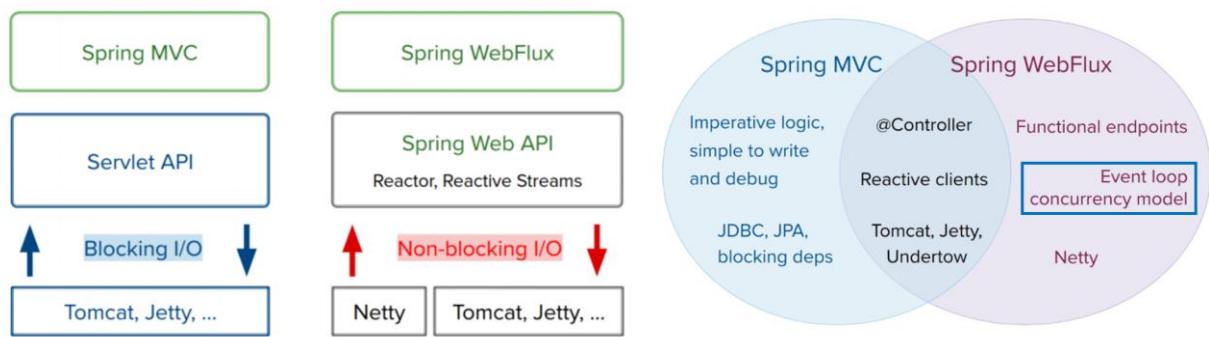
### Что же такое WebFlux?

Это асинхронный и неблокирующий микро-фреймворк, который позволяет одновременно обрабатывать огромное количество соединений.

Модуль WebFlux является альтернативой Spring MVC и представляет собой **реактивный** подход для написания веб-сервисов.

Он построен на библиотеке Reactor и работает, начиная с версии SpringBoot 2 и выше. По умолчанию он использует Netty (не Tomcat).





Spring Framework поддерживает реактивное программирование, начиная с пятой версии. Эта поддержка построена на основе Project Reactor.

Project Reactor или просто **Reactor** — это библиотека Reactive для создания неблокирующих приложений на JVM, основанная на спецификации **Reactive Streams**.

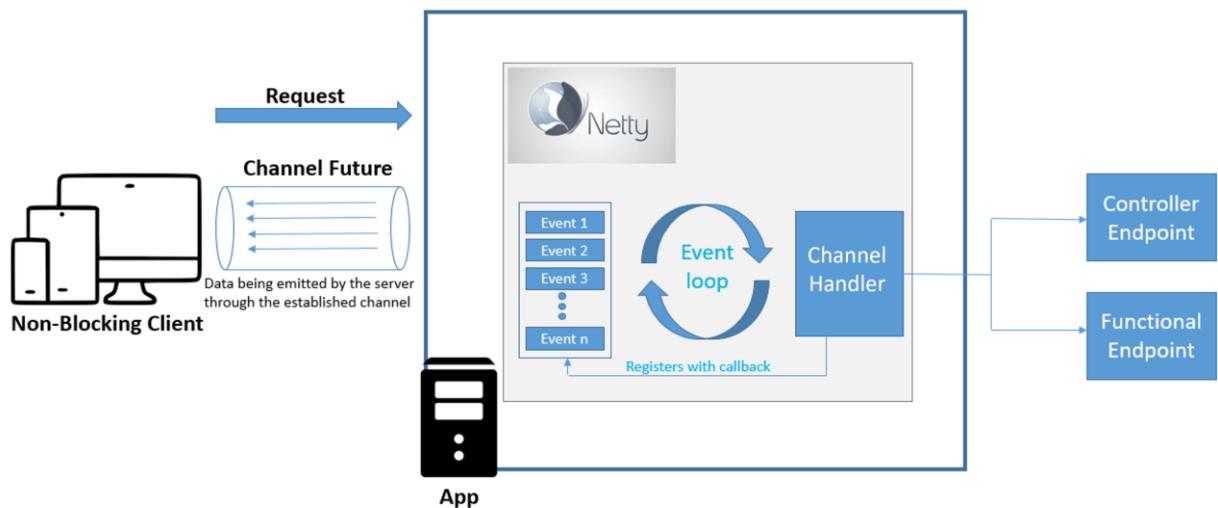
**Это основа реактивного стека в экосистеме Spring.**

Spring **WebFlux** использует модель обработки запросов **EventLoop** (цикл событий).

Отличное видео, пусть вас не смущает js (модель универсальна) → <https://youtu.be/8aGhZQkoFbQ>

Когда на сервер Netty приходит запрос, он немедленно отвечает объектом Future, таким образом, не блокируя клиента. Тем временем он может делать все, что хочет.

Бонусом является то, что Netty обрабатывает большое количество соединений. **События – это и запросы Request, и ответы Callback, которые регистрирует Netty.**

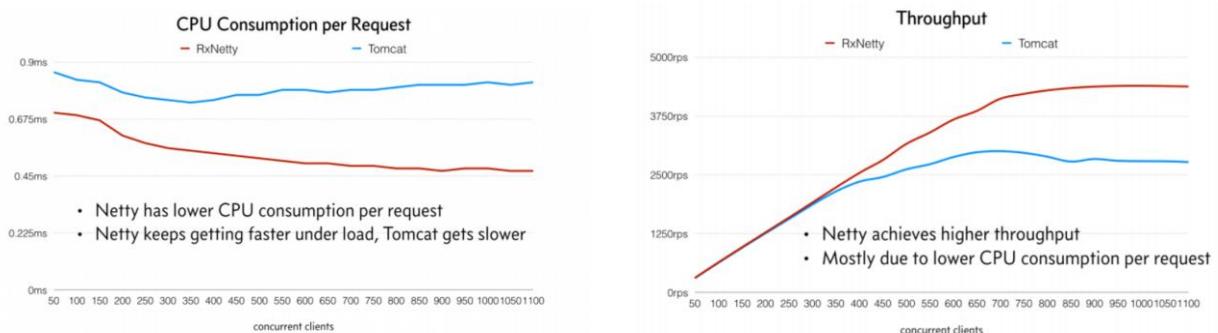


Соединение между клиентом и сервером устанавливается через канал. **EventLoop** Netty ищет события. У него есть **внутренняя очередь**, в которой он хранит события и Цикл Событий, принимающий в порядке FIFO (первый вошёл, первый вышел), до тех пор, пока он не обработает их все.

**ChannelHandler** ищет правильную конечную точку для вызова: либо **Controller Endpoint** (в классах, аннотированных **@RestController**), либо **Functional Endpoint** (функциональная конечная точка). Цикл событий Netty связывает каждый запрос с каналом, и таким образом он знает, на какой канал ему нужно отправить ответ.

Посмотрим, сколько вычислительных ресурсов расходуют Netty и Tomcat на выполнение одного запроса. Throughput — это общее количество обработанных данных.

При небольшой нагрузке, до первых 300 пользователей у RxNetty и Tomcat оно одинаковое, а после **Netty** (красная линия) **уходит в отрыв — почти в 2 раза!**



### Как вызвать транзакционный метод из того же класса?

В Spring Framework существует аннотация `@Transactional`. Ей помечается метод или класс, весь код которого должен выполняться в рамках транзакции. Обычно имеется в виду транзакция базы данных, но вообще это **понятие определяется используемым transactionManager-ом**. Настройки, такие как уровень изоляции, стратегия роллбэка и прочие, определяются через параметры этой аннотации.

В теории, `@Transactional` делает метод транзакционным для этого класса и всех его наследников. На практике же, по умолчанию, если вызвать транзакционный метод `Foo.bar()`, из `Foo.baz()`, то транзакция не создастся.

Это происходит вследствие того, что по умолчанию Spring AOP добавляет код открытия и закрытия транзакции через динамический proxy класс. То есть, вместо `Foo` инжектится нечто, похожее на изображении.

Первый вариант решения проблемы – вместо аннотации использовать `TransactionTemplate`, то есть обернуть код в транзакцию вручную.

Примеры использования в статье: <https://www.baeldung.com/spring-programmatic-transaction-management>

Другой, более универсальный, но более сложный в конфигурации способ – переключить режим работы Spring AOP с динамических прокси на нечто другое. Обычно применяется библиотека AspectJ:

```
@EnableTransactionManagement(mode = AdviceMode.ASPECTJ)
```

В Spring AOP есть понятие **weaving** – этап добавления дополнительной функциональности (аспектов).

В нашем случае, это код открытия и закрытия транзакции.

Чтобы заработал weaving AspectJ этапа компиляции, в сборку нужно добавить плагин: `aspectj-maven-plugin` для maven, `gradle-aspectj` для gradle.

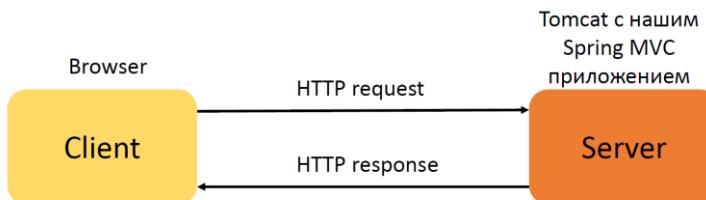
```
class GeneratedProxyForFoo extends Foo {  
    @Autowired PlatformTransactionManager manager;  
  
    @Override void bar() {  
        var tx = manager.getTransaction(annotationConfig);  
        try {  
            super.bar();  
            manager.commit(tx);  
        } catch (Exception e) {  
            manager.rollback(tx);  
        }  
    }  
}
```

\*Когда отрабатывает `@Autowired` а когда `@Transactional`? почему?  
Почему нельзя отработать `@Transactional` в другом методе?

`@Autowired` работает в методе ДО инициализации, `@Transactional` после.  
Так как оборачивает логику в прокси.

## Протокол HTTP

HTTP – это протокол, который используется для передачи данных в сети



## Структура HTTP запроса, ответа

Структура протокола определяет, что каждое HTTP-сообщение состоит из трёх частей, которые передаются в следующем порядке:

**Стартовая строка** (англ. Starting line) — определяет тип сообщения;

**Заголовки** (англ. Headers) — характеризуют тело сообщения, параметры передачи и прочие сведения;

**Тело сообщения** (англ. Message Body) — непосредственно данные сообщения.  
Обязательно должно отделяться от заголовков пустой строкой.

```

No. | Source | Destination | Protocol | Info
31 | 192.168.3.1 | 194.188.210.1 | HTTP | HTTP/1.0 302 Moved Temporarily
37 | 192.168.3.1 | 194.188.210.1 | HTTP | HTTP/1.0 200 OK (text/html)
54 | 192.168.3.1 | 194.188.210.1 | HTTP | HTTP/1.0 200 OK (text/css)

HTTP/1.0 200 OK\r\n
Server: Apache/2.2.3 (CentOS)\r\n
Last-Modified: Wed, 09 Feb 2011 17:13:11 GMT\r\n
Content-Type: text/html; charset=UTF-8\r\n
Accept-Ranges: bytes\r\n
Date: Thu, 03 Mar 2011 04:04:36 GMT\r\n
Content-Length: 2945\r\n
Age: 13165\r\n
X-Cache: HIT from proxy.cmgtu (squid/3.1.8)\r\n
Via: 1.0 proxy.cmgtu (squid/3.1.8)\r\n
Connection: keep-alive\r\n
\r\n
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/HTML/DTD/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>IANA &mdash; Example domains</title>
<!-- start common-head -->
<meta http-equiv="Content-type" content="text/html; charset=utf-8" />
<link rel="stylesheet" type="text/css" href="/_css/reset-fonts-grids.css" />
<link rel="stylesheet" type="text/css" media="screen" href="/_css/screen.css" />
<link rel="stylesheet" type="text/css" media="print" href="/_css/print.css" />
<link rel="shortcut-icon" type="image/ico" href="/favicon.ico" />
</head>
<body>
<h1>IANA &mdash; Example domains</h1>
<ul>
<li>0000 48 54 54 50 2f 31 2e 30 20 32 38 38 20 4f 4b 0d</li>
<li>0010 0e 53 65 72 76 65 72 3a 2e 41 70 61 63 68 65 2f</li>
</ul>
</body>
</html>

```

## HTTP методы

REST API основаны на HTTP-запросах или командах, каждая из которых выполняет свою задачу.

REST поддерживает следующие HTTP-запросы:

Метод GET : запрос данных с сервера.

Метод POST : отправьте данные для создания нового ресурса по указанному сервером URL-адресу.

Метод PUT : отправьте данные для создания нового ресурса по указанному клиентом URL-адресу.

Метод DELETE : удалить ресурс с сервера.

Метод OPTIONS : возвращать методы запроса, поддерживаемые службой.

Метод HEAD : возвращает метаинформацию, такую как заголовки ответа.

Метод PATCH : изменить часть ресурса на сервере.

## Ключевое отличие методов POST и GET

## HTTP протокол

@GetMapping связывает HTTP запрос, использующий HTTP метод GET с методом контроллера.

@PostMapping связывает HTTP запрос, использующий HTTP метод POST с методом контроллера.

GET

POST

- Передаваемая информация хранится в URL
  - Ограничен максимальной длиной
  - Не поддерживает передачу бинарных данных
  - Можно поделиться ссылкой
  - Как правило, используется для получения информации
- Передаваемая информация хранится в теле запроса
  - Не ограничен максимальной длиной
  - Поддерживает передачу бинарных данных
  - Ссылкой поделиться нельзя
  - Как правило, используется для добавления данных

HTTP метод	URL	CRUD операция
GET	api/employees	Получение всех работников
GET	api/employees/{employeeId}	Получение одного работника
POST	api/employees	Добавление работника
PUT	api/employees	Изменение работника
DELETE	api/employees/{employeeId}	Удаление работника

### REST стиль

**REST API (Representational State Transfer API)** – это архитектурный стиль взаимодействия компонентов распределенного веб-приложения.

Компоненты в REST взаимодействуют в сети Интернет в качестве **клиентов и серверов**. REST - набор соглашений взаимодействия с HTTP.

Определяет стиль взаимодействия между разными компонентами системы.

**REST означает передачу репрезентативного состояния.**

Если разрабатывается публичное API и логика взаимодействия во многом покрывается четверкой методов CRUD - выбирается REST. Он наиболее популярен в WEB. Яндекс, Google и другие используют именно его для своего API.

**ВАЖНО!** Корзина должна быть на стороне клиента (иначе это не REST).

Храним состояние клиента на стороне КЛИЕНТА (а не на сервере).

Клиент наполняет корзину и ОДИН раз id заказа отправляем на сервер.

Representational - ресурсы в REST могут быть представлены в любой форме - JSON, XML, текст, или даже HTML- зависит от того, какие данные больше подходят потребителю.

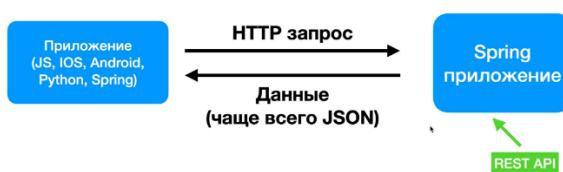
**Настоящий RESTful API должен соответствовать ПЯТИ архитектурным ограничениям REST:**

- 1) Единый интерфейс - необходим для отделения клиента от сервера.
- 2) Клиент-сервер
- 3) Без гражданства - Отсутствие состояния означает, что клиент и сервер не хранят информацию о состоянии друг друга. Поскольку **сервер не хранит никакой информации**, он обрабатывает каждый запрос клиента как новый запрос.
- 4) Кэшируемый
- 5) Многоуровневая система

Код по запросу — это необязательное ограничение архитектуры RESTful.

Код по запросу позволяет серверу отправлять исполняемый код (скрипты или апплеты) клиенту по запросу клиента.

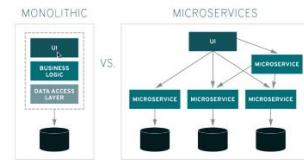
## Самый частый сценарий использования Spring Framework



Такие приложения, которые отдают данные в обиходе называются REST API (синонимы: RESTful API, сервис, бэкенд, API).

## REST API очень популярен

Отчасти благодаря популярности микросервисной архитектуры приложений



Когда приложение разрастается, его становится сложно изменять, дополнять и масштабировать, поэтому его делят на небольшие, слабо связанные и легко изменяемые приложения, которые выполняют относительно элементарные функции.

Эти автономные приложения называются **микросервисами**.

Микросервисы общаются друг с другом с помощью API. Совокупность микросервисов, общающихся друг с другом с помощью API, формирует итоговое приложение.

### REST API

Вызовы REST API могут осуществляться, используя HTTP протокол

REST API Не принуждает использовать какой-то конкретный язык программирования.  
Клиентская и серверная части нашего приложения могут быть написаны на разных языках программирования

Для передачи информации можно использовать не только JSON, но и любой другой формат данных

### JSON

#### JSON – JavaScript Data Notation

Формат данных JSON представляет собой текстовую информацию

JSON используется для хранения и особенно для обмена информацией

JSON не привязан к какому-то конкретному языку программирования и используется повсеместно

Формат данных JSON содержит коллекцию пар ключ-значение

JSON Data Binding или JSON Mapping – это привязка JSON к Java объекту

## Идемпотентность: что это такое, методы

### Понимание идемпотентности.

Примером идемпотентной операции может быть **операция умножения числа на единицу**. Независимо от того, сколько раз вы умножаете пять на один, вы получите тот же результат.

### Результат может меняться, но состояние НЕТ.

**Безопасный запрос** - это запрос, который не меняет состояние приложения.

**Идемпотентный запрос** - это запрос, эффект которого от многократного выполнения равен эффекту от однократного выполнения.

метод	безопасный	идемпотентный
get	да	да
put	нет	да
delete	нет	да
post	нет	нет
options	да	да

#### ИДЕМПОТИЧНОСТЬ:

Сколько бы раз не применили операцию - результат будет тем же.

Как если бы применили один раз.

Пример: умножение на единицу. Сколько не повторяй - результат будет тем же

**POST** - не идемпотентен, остальные - Да! Пример - email (если один адрес уже создали, то другой не создать)

Говоря простым языком, сколько бы раз вы не применяли операцию, её результат будет один и тот же, как если бы вы применили её всего один раз.

Например, вы поздоровались с утра с человеком, сказав ему «Привет!» В результате ваш знакомый переходит в состояние «поздорованный» :-)

И если вы ещё несколько раз в течение дня ему скажете «Привет!», ничего не изменится, он останется в том же состоянии.

**ВАЖНО!**

Идемпотентность не гарантирует. Что получим такой же ответ.

Например, используя метод DELETE, первый раз объект удалили, а второй раз получили 404 not found... И, хотя состояние объекта не изменилось, но получили другой ответ.\

То есть ответ может меняться в зависимости от операции.

**Как метод POST сделать идемпотентным?** Сделать метод PUT и настроить проверку.

Лучше разобраться в понятиях экосистемы Spring поможет **известный доклад Евгения Борисова «Spring-потребитель»** (2 части): <https://www.youtube.com/watch?v=BmBr5diz8WA> [https://www.youtube.com/watch?v=cou\\_qomYLNU](https://www.youtube.com/watch?v=cou_qomYLNU)

**Как все устроено?** (краткое содержание)

В 2003 году появилась возможность настраивать контекст при помощи XML (конфиг файла). Там прописываем бины, потом поднимаем контекст и так это работает.

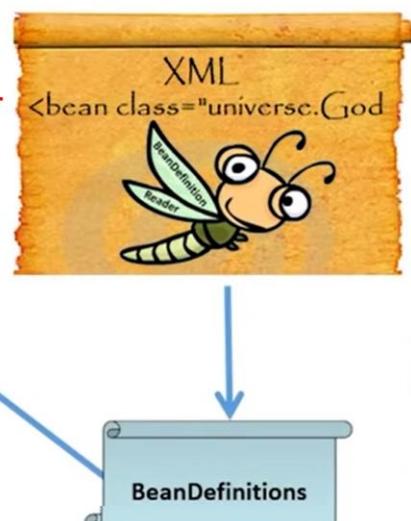


**XmlBeanDefinitionReader** – это класс, внутренний компонент Spring, который сканирует XML файл и переводит то, что мы написали в **BeanDefinition** – объекты, которые хранят в себе инфу про бины.



**BeanFactory** – это интерфейс, центральный игрок, отвечает за создание и хранение всех объектов (синглтонов).

**BeanFactory** – это реализация паттерна Фабрика, его функциональность покрывает создание бинов. Так как эта фабрика знает многое об объектах приложения, то она может создавать связи между объектами на этапе создания экземпляра. Существует несколько реализаций **BeanFactory**, самая используемая – "org.springframework".

**Декларация:**

1. Из какого класса бин создавать.
2. Есть ли Init метод?
3. Если «да», то как он называется
4. Какие property и т.д.

static	7	<bean class="kata.spring_borisov.quoters.TerminatorQuoter" id="terminatorQuoter">
templates	8	<property name="message" value="I'll be back"/>
application.properties	9	</bean>
context.xml		

```
INFO 4736 --- [           main] k.s.SpringBorisovApplication      : Started SpringBorisovApplication
3.289 seconds (process running for 5.023)
message = I'll be back
```



**BeanPostProcessor** позволяет настраивать бины **ДО** того, как они попадают в контейнер.

Задействован паттерн **Chain of responsibility** (цепочка обязанностей) – поведенческий шаблон проектирования, предназначенный для организации в системе уровней ответственности.

Интерфейс **BeanPostProcessor** позволяет **вклиниваться** в процесс настройки бинов до того, как они попадут в контейнер. У интерфейса два важных метода.

Оба вызываются для каждого бина:

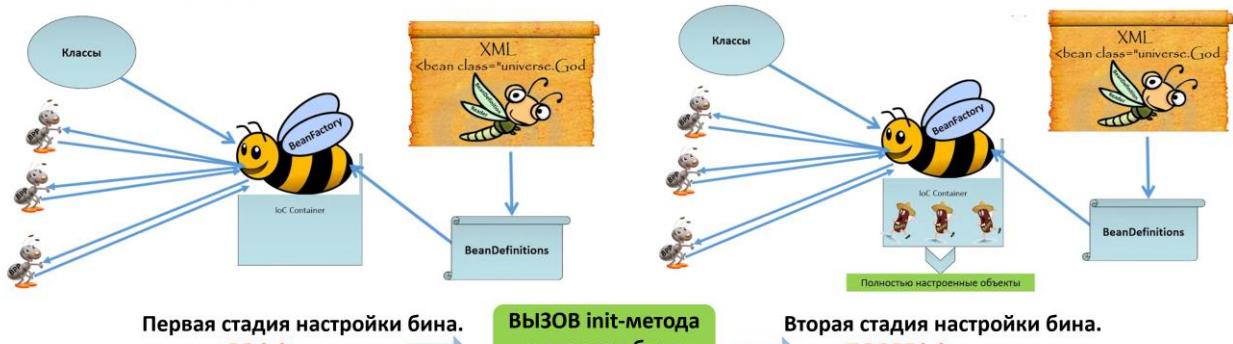
- `postProcessBeforeInitialization (Object bean, String beanName)`
- `postProcessAfterInitialization (Object bean, String beanName)`

**А между ними вызывается init-метод**

- **init-метод** (если работаем с XML, то можем прописать init-метод в теге bean)

```
<bean class="kata.spring_borisov.quoters.InjectRandomIntAnnotationBeanPostProcessor" />
```

- **@PostConstruct** (если работаем с аннотациями)
- **AfterPropertiesSet** (в Spring 2 имплементируем интерфейс **InitializingBean** и прописываем этот метод AfterPropertiesSet) – так давно уже никто не делает



1. **XmlBeanDefinitionReader**, считывает все декларации бинов и кладёт их в Мар.
2. **BeanFactory** – фабрика бинов получает информацию и в первую очередь настраивает бины, которые имплементируют интерфейс **BeanPostProcessor** (потом с их помощью будет настраивать остальные).
3. После того, как BF настроила бины согласно конфигурации, BF отдаёт бин BPP для до-настройки.
4. Запуск init-метода.

1. После того, как BF получила бины от BPP (`postProcessBeforeInitialization`)
2. **Запуск init-метода**
3. BF отдаёт бин BPP для **дополнительной настройки** (`postProcessAfterInitialization`)

За обработку **@Autowired** отвечает **BeanPostProcessor** (соответствует этапу ЖЦ бина)!

**Кейс 1: создаём пользовательскую аннотацию для внедрения.**

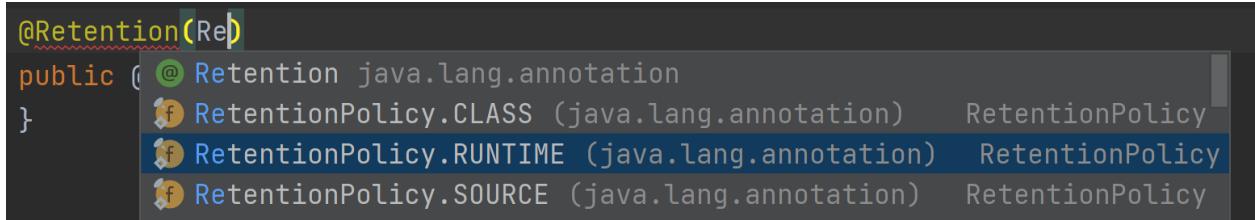
Например, если мы хотим «кастомизировать» Spring приложение, в котором нужна функция генерации случайных чисел.

И задача не просто генерировать рандом, а добавлять эти значения в поля классов.

Сделаем это декларативно через **кастомизированную (пользовательскую) аннотацию**. Мы будем ставить ее НАД теми полями, в которые нужно внедрить рандомное число. А потом научим Spring после создания бина ДОнастраивать его в соответствии с нашей авторской аннотацией.

Шаг 1: создаём пользовательскую аннотацию `@InjectRandomInt(min = 2, max = 7)`

Шаг 2: создастся интерфейс аннотаций (public @interface), ставим `@Retention`, которая указывает, КАК долго должны сохраняться аннотации с аннотированным интерфейсом.



По умолчанию `CLASS` – это аннотация в байт код попасть должна, но через `Reflection` и `runtime` считать ее не сможем, ее там не будет. Аннотации должны быть записаны компилятором в файл класса, но не должны сохраняться виртуальной машиной во время выполнения. **Это поведение по умолчанию**.

`SOURCE` видна только в ресурсах. Аннотации должны быть отброшены компилятором. Когда код компилируется, то в байт коде уже ничего не будет (например, `@Override`, если его поставим там, где это не логично, то компилятор будет ругаться).

**Выбираем `RUNTIME`** – аннотации должны быть записаны компилятором в файл класса и сохранены виртуальной машиной во время выполнения, чтобы их можно было читать рефлексивно.

ШАГ 3: создаём класс с длинным названием 😊 и имплементируем интерфейс `BeanPostProcessor` + переопределяем 2 метода. В методе `postProcessBeforeInitialization` прописываем логику, которая должна быть внедрена ДО инициализации бина.

**Кейс 2: создаём профилирующую аннотацию, изменяющую поведение бина**

**Задача:** сделать так, чтобы все классы, над которыми стоит аннотация `@Profiling` профилировались (будет выведено время работы методов).

Технически это **BeanPostProcessor**, который будет к этой аннотации относиться.

Алгоритм работы ВРР для решения задачи:

- получает бин от BF
- проверяет, не стоит ли над этим бином аннотация `@Profiling`
- и если стоит, то дописывает в каждый метод этого бина логику, связанную с профайлингом (замерить время ДО, время ПОСЛЕ и вывести в лог разницу t)

Т.к. будет модификация уже созданного объекта, то это можно решить двумя способами:

- Наследование + переопределение методов с добавлением нужной логики – **cjlib**
- Имплементация тех же самых интерфейсов – динамический **proxy**

У наследования есть ограничения (не от любого класса можно наследоваться, например, `final`). Поэтому Spring предпочитает идти через интерфейсы.

По такому же принципу работает АОР (через прокси). И если Spring аспекту нужно сделать прокси на какой-нибудь объект, то он сначала смотрит, есть ли у него интерфейсы? Если да, идет через динамический прокси. А если нет, то идет через `cjlib`.

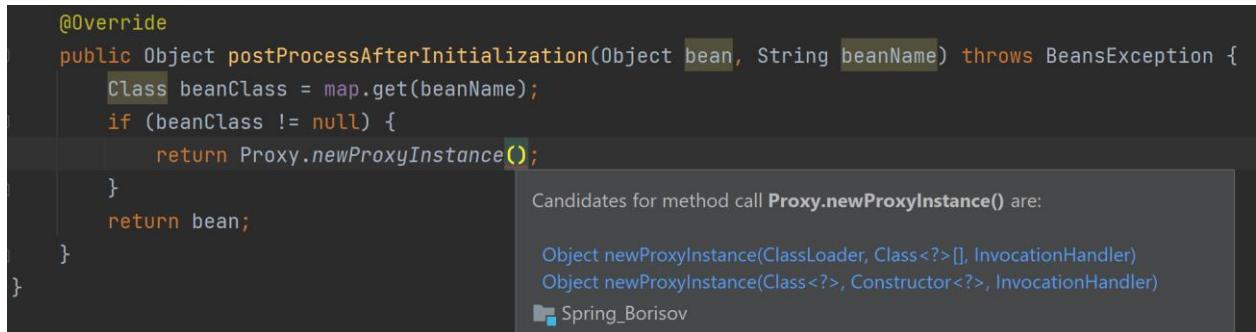
По конвенциям Spring те ВРР, которые что-то меняют в классе, должны это делать на этапе `postProcessAfterInitialization` (после `init`-метода). Таким образом мы точно знаем, что `@PostConstruct` всегда работает на **оригинальный метод** (до того, как все proxy накрутились на него).

Шаг 1: создаём пользовательскую аннотацию @Profiling над классом

Шаг 2: создаём класс с длинным названием ProfilingHandlerBeanPostProcessor, имплементируем интерфейс BeanPostProcessor, переопределяем методы.

ШАГ 3: на этапе ДО init-метода откладываем в сторону те бины, поведение которых хотим изменить: `private Map<String, Class> map = new HashMap<>();`

ШАГ 4: на этапе ПОСЛЕ init-метода проверяем, есть ли в мапе имя этого бина. Статический метод `newProxyInstance()` создаёт объект из НОВОГО класса, который будет сгенерирован им же самим динамически (на лету).



```
@Override
public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
    Class beanClass = map.get(beanName);
    if (beanClass != null) {
        return Proxy.newProxyInstance();
    }
    return bean;
}
```

#### Параметры метода:

`loader` - загрузчик класса для определения прокси-класса

То есть `ClassLoader` загрузит класс, который будет сгенерирован на лету (динамически) и класс загрузится в Permanent область памяти (PermGen). А начиная с Java 8 в HEAP. <http://www.javaspecialist.ru/2011/04/permanent.html>

`interfaces` - список интерфейсов для реализации прокси-класса

`handler` - обработчик вызова для отправки вызовов метода. **Invocation Handler** – это такой объект, который инкапсулирует логику, которая попадёт во все методы класса, который сгенерируется на лету)))

**Метод возвращает** экземпляр прокси с указанным обработчиком вызова прокси-класса, который определен указанным загрузчиком класса и который реализует указанные интерфейсы.

**ApplicationListener** – это функциональный интерфейс, реализуемый слушателями событий приложения. Он умеет слушать контекст Spring, все события, которые происходят.

Это функциональный интерфейс, поэтому его можно использовать в качестве цели назначения для лямбда-выражения или ссылки на метод. На основе стандартного интерфейса `java.util.EventListener` для шаблона проектирования **Наблюдатель (Observer)**.



Начиная с Spring 3.0, `ApplicationListener` может в общем случае объявить интересующий его тип события.

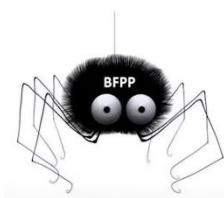
При регистрации в `Spring ApplicationContext` события будут соответствующим образом фильтроваться, а слушатель будет вызываться только для сопоставления объектов событий. Из любого события можно вытащить контекст.

`ContextStartedEvent` (начал построение бинов)

`ContextStoppedEvent` (остановился)

`ContextRefreshedEvent` (закончил построение бинов)

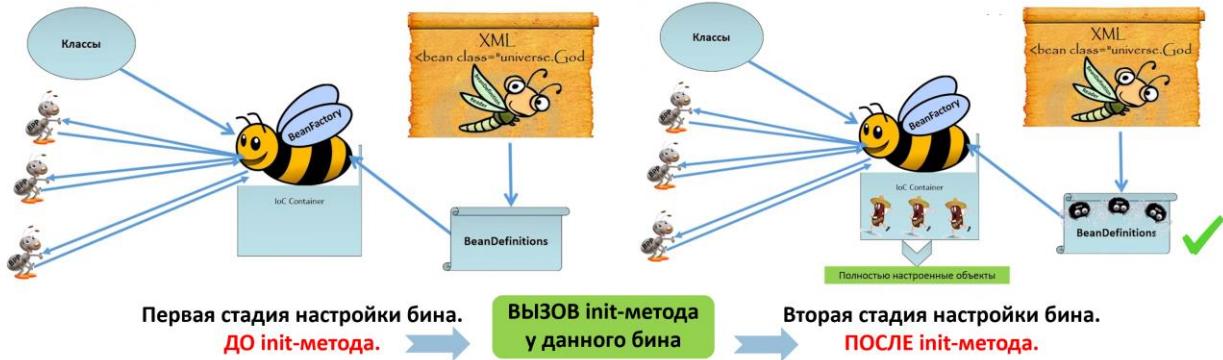
`ContextClosedEvent`



**BeanFactoryPostProcessor** – позволяет настраивать BeanDefinition до того, как создаются бины.

Интерфейс имеет один единственный метод `PostProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)`.

Этот метод запустится на этапе, когда другие бины еще не созданы и есть только BeanDefinition -s и сам BeanFactory.



1. **XmlBeanDefinitionReader**, считывает все декларации бинов и кладёт их в Мап.
2. **BeanFactory** – фабрика бинов получает информацию и в первую очередь настраивает бины, которые имплементируют интерфейс **BeanPostProcessor** (потом с их помощью будет настраивать остальные).
3. После того, как BF настроила бины согласно конфигурации, BF **отдаёт бин BPP для до-настройки**.
4. **Запуск init-метода.**

1. После того, как BF получила бины от BPP (`postProcessBeforeInitialization`)
2. **Запуск init-метода**
3. **BeanFactoryPostProcessor** –ы (паучки) доработали декларации и обновили их
4. Создались BeanPostProcessor –ы (BPP муравьи)
5. BF отдаёт бин BPP для **дополнительной настройки** (`postProcessAfterInitialization`)
6. **BF кладёт готовые бины в IoC контейнер. ГОТОВО!**

@Component

Конфигурация через аннотации  
«под капотом»

- `<context:component-scan base-package="com..." />`
- `new AnnotationConfigApplicationContext("com")`
- ClassPathBeanDefinitionScanner
- Не является ни BeanPostProcessor-ом, ни BeanFactoryPostProcessor-ом
- Он ResourceLoaderAware
- Создаёт BeanDefinitions из всех классов, над которыми стоит `@Component`, или другая аннотация, аннотированная `@Component`



Java Config

- `new AnnotationConfigApplicationContext(JavaConfig.class);`
- Казалось бы, его должен парсировать, какой-нибудь BeanDefinitionReader, как это было с XML
- И даже его класс его называется скоже: AnnotatedBeanDefinitionReader.
- Но нет, AnnotatedBeanDefinitionReader вообще ничего не имплементирует
- Он просто является частью ApplicationContext-а
- Он только регистрирует все JavaConfig-и

## Определение бина в Java коде.

Это Java файл, на него с помощью CJLib будет накручен PROXY.

При вызове бина будет делегация в метод, который прописан в конфиге.

```
@Configuration // конфигурация
@ComponentScan("root") // какой пакет сканировать
public class JavaConfig {
    @Bean // название бина
    public CoolDao dao() {
        return new CoolDaoImpl();
    }

    @Bean(initMethod = "init") // название init-метода
    @Scope(BeanDefinition.SCOPE_PROTOTYPE) // область видимости бина
    public CoolService coolService() { // название бина
        CoolServiceImpl service = new CoolServiceImpl();
        service.setDao(dao());
        return service;
    }
}
```

Это не метод в классическом понимании.  
Это ОПИСАНИЕ БИНА в формате метода, где описана логика, которую нужно будет каждый раз запускать, когда этот бин будет создаваться.

## Кто обрабатывает JavaConfig?

- ConfigurationClassPostProcessor (особый BeanFactoryPostProcessor)
- Его регистрирует AnnotationConfigApplicationContext
- Он создаёт бин-дифинишины по @Bean
- А так же относится к:
  - @Import
  - @ImportResource
  - @ComponentScan (да-да, там опять будет задействован крот)