**Введение в Spring**

Введение в Spring. Dependency Injection и Inversion of Control. Бины. Spring Context. Конфигурирование Spring (XML, Java Config, аннотации).

**Оглавление**

[Что такое Spring?](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.3znysh7)

[Внедрение зависимостей и инверсия управления](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.2et92p0)

[Каким образом Spring облегчает разработку?](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.tyjcwt)

[Конфигурирование Spring](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.3dy6vkm)

[Конфигурирование с использованием XML](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.1t3h5sf)

[Конфигурирование с использованием аннотаций](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.4d34og8)

[JavaConfig](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.4g14xtnty7ow)

[Советы в выборе способа конфигурации](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.3rdcrjn)

[Внедрение зависимостей в Spring](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.26in1rg)

[Внедрение примитивных типов](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.lnxbz9)

[Внедрение объектов](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.35nkun2)

[Область видимости бинов](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.grz4z51q79bu)

[Этапы инициализации контекста](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.pa2g6bdak70m)

[Этап 1](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.14qirpis3nq0)

[Этап 2](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.u65qzkx0qm8w)

[Этап 3](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.wnrmri97395f)

[Этап 4](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.ggid5kgd7w56)

[Жизненный цикл бина](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.kajzv9ygo5t9)

[Практическое задание](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.1ksv4uv)

[Дополнительные материалы](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.44sinio)

[Используемая литература](https://docs.google.com/document/d/1mcNneG9V4gjiPI6UV6e3Tj1sQFhKiMve9WWVcFkipSA/edit#heading=h.2jxsxqh)

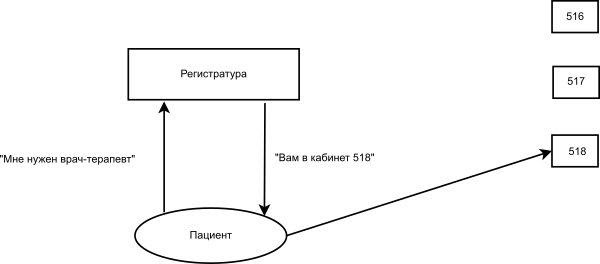
**Что такое Spring?**

Spring — один из самых популярных фреймворков для разработки веб-приложений на языке Java, главным преимуществом которого является простота и легковесность. Фреймворк Spring представляет собой контейнер Inversion of Control (IoC).

**Внедрение зависимостей и инверсия управления**

Inversion of Control (IoC, инверсия управления) — обозначение важного принципа ООП, согласно которому контроль над управлением элементом программы (например, отдельным классом) передается от вас отдельному компоненту. Форм такого контроля может быть множество, но Spring реализует одну из них — внедрение зависимостей. Выясним, зачем доверять управление классами отдельному компоненту и что-то внедрять.

В повседневной жизни много примеров инверсии управления и внедрения зависимостей. Вы приходите в больницу на прием к врачу, и, чтобы узнать номер кабинета, обращаетесь в регистратуру. На основе тех данных, которые получит от вас регистратор (например, «мне нужен терапевт»), вам дадут информацию. Конечный выбор номера кабинета остается не за вами, а за регистратором, — произошла инверсия управления. Да и зачем самостоятельно искать нужный кабинет? Вы знаете только то, что вам нужно к определенному специалисту, который выполняет конкретные функции (а значит, реализует определенный интерфейс). Графически этот процесс представлен на рисунке:



Плюсы от такого подхода в реальной жизни очевидны.

Разберемся с инверсией управления в коде. Представим, что должны написать программу, которая умеет делать фото с помощью пленочного фотоаппарата.  Нужно разработать два класса: **Camera** (фотоаппарат) и **CameraRoll** (фотопленка). Учитывая, что фотопленка «заряжается» в фотоаппарат, наш класс **Camera** будет содержать поле типа **CameraRoll**.

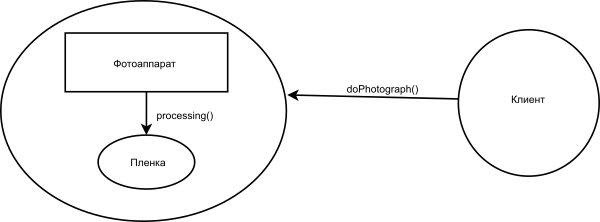
Класс **Camera** будет иметь следующий вид:

|  |
| --- |
| public class Camera {    private CameraRoll cameraRoll;      public void doPhotograph() {      System.out.println("Щелк!");      cameraRoll.processing();    }  } |

Ничего необычного в этом классе не происходит. При вызове метода **doPhotograph()** имитируется процесс создания фотографии (слышен звук щелчка), а с помощью метода **processing()** происходит отработка кадра на пленке. Класс **CameraRoll** будет выглядеть так:

|  |
| --- |
| public class CameraRoll {    public void processing() {      System.out.println("-1 кадр");    }  } |

Таким образом имитируется следующее отношение:



Все выглядит вполне просто. При чем здесь инверсия управления и внедрение зависимостей? Попробуем сделать фотографию и напишем следующий клиентский код:

|  |
| --- |
| public class Client {    public static void main(String[] args) {      Camera camera = new Camera();      camera.doPhotograph();    }  } |

В данном коде мы создаем объект фотоаппарата и пытаемся сделать фотографию. Что в итоге? Слышим щелчок, но фото не получаем. Чтобы сделать снимок, необходима пленка внутри фотоаппарата. Изменим его код, ведь он полностью зависит от фотопленки и без нее работать не будет.

Первое, что сделали бы некоторые:

|  |
| --- |
| public class Camera {    private CameraRoll cameraRoll = new CameraRoll();    public void doPhotograph(){      System.out.println("Щелк!");      cameraRoll.processing();    }  } |

Фотоаппарат заработал, но где вы видели камеры, которые производятся со встроенной пленкой? И зачем так увязывать фотоаппарат с конкретной фотопленкой? Перепишем иначе:

|  |
| --- |
| public class Camera {    private CameraRoll cameraRoll;      public Camera(CameraRoll cameraRoll) {      this.cameraRoll = cameraRoll;    }      public void doPhotograph() {      System.out.println("Щелк!");      cameraRoll.processing();    }  } |

Все будет работать. В данном листинге мы наблюдаем внедрение зависимости через конструктор. Да, мы сделали предыдущий код более гибким, и теперь сами выбираем, какой объект класса фотопленки вставить в фотоаппарат. Но получается, что, создав фотоаппарат единожды и применив при этом любую понравившуюся нам фотопленку, мы больше не сможем поменять ее, даже когда она закончится. А хотелось бы иметь возможность вставлять в аппарат и доставать из него любую фотопленку сколько угодно раз. Используем для этого **get** и **set**. Перепишем код:

|  |
| --- |
| public class Camera {    private CameraRoll cameraRoll;    public CameraRoll getCameraRoll() {      return cameraRoll;    }    public void setCameraRoll(CameraRoll cameraRoll) {      this.cameraRoll = cameraRoll;    }    public void doPhotograph(){      System.out.println("Щелк!");      cameraRoll.processing();    }  } |

Теперь можем сами менять фотопленку в любое время. А чтобы использовать черно-белую пленку, необходимо написать ее интерфейс и две реализации: цветную и черно-белую.

Интерфейс и две его реализации будут содержать следующий код:

|  |
| --- |
| public interface CameraRoll {    public void processing();  }  public class ColorCameraRoll implements CameraRoll {    @Override    public void processing() {      System.out.println("-1 цветной кадр");    }  }  public class BlackAndWhiteCameraRoll implements CameraRoll {    public void processing() {      System.out.println("-1 черно-белый кадр");    }  } |

С первого варианта кода до данного момента мы «отвязали» фотоаппарат не просто от конкретного объекта класса пленки, но и от конкретного вида самой пленки. Все это получилось благодаря использованию get- и set-метода и интерфейса (внедрение через сеттер).

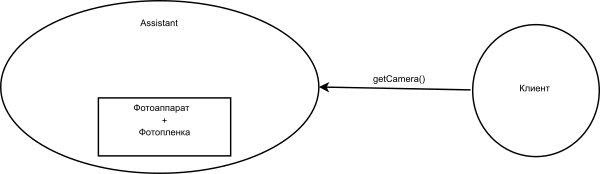
Теперь фотоаппарат умеет работать с несколькими видами пленки, и менять мы ее можем в любое время.

Взглянем теперь, какие действия необходимо произвести клиенту, чтобы сделать фотографию. Клиентский код будет иметь следующий вид:

|  |
| --- |
| public class Client {    public static void main(String[] args) {      Camera camera = new Camera();      CameraRoll cameraRoll = new ColorCameraRoll();      camera.setCameraRoll(cameraRoll);      camera.doPhotograph();    }  } |

Фотоаппарат теперь отлично настраивается. В этом коде присутствуют вполне шаблонные действия: создание объекта фотоаппарата, фотопленки, внедрение зависимости. Но зачем клиенту заботиться об этом? Ему просто нужно сделать фотографию, а созданием и внедрением пусть занимается тот, кто в дальнейшем отдаст настроенный фотоаппарат клиенту, чтобы он сделал фото.

В идеале это должно выглядеть следующим образом:



И клиентский код теперь выглядит так:

|  |
| --- |
| public class Client {    public static void main(String[] args) {      Assistant assistant = new Assistant();      Camera camera = assistant.getCamera();      camera.doPhotograph();    }  } |

Теперь у клиента есть помощник, который делает всю второстепенную работу за него. Он управляет процессом создания и настройки фотоаппарата — произошла инверсия управления. Основной функцией помощника является вставка фотопленки в фотоаппарат — это основная форма инверсии управления, которая называется внедрением зависимости. В данном случае класс **Assistant** является для нас подобием IoC-контейнера. Здесь и кроется идея Spring!

**Каким образом Spring облегчает разработку?**

Spring является реализацией IoC-контейнера, а также предоставляет дополнительные службы для содержащихся в нем объектов. Подобный контейнер состоит из двух компонентов:

* **контекста**, который хранит созданные объекты с уже внедренными зависимостями (в нашем примере — объект фотоаппарата);
* **дополнительных служб**, благодаря которым создаются объекты и внедряются зависимости. Но есть и множество других служб, которые определяют функциональность Spring. Главная особенность фреймворка в том, что он реализует модульную архитектуру, где каждый модуль представляет собой JAR-файл и наделен определенной функциональностью. Можно подключать только те модули, которые нужны для решения конкретной задачи.

Для начала работы со Spring необходимо создать пустой Maven-проект и подключить зависимости в **pom.xml**. Все зависимости будем подключать c mvnrepository.com.

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->  <dependency>      <groupId>org.springframework</groupId>      <artifactId>spring-core</artifactId>      <version>5.0.8.RELEASE</version>  </dependency>  <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->  <dependency>      <groupId>org.springframework</groupId>      <artifactId>spring-context</artifactId>      <version>5.0.8.RELEASE</version>  </dependency> |

Spring-core — это основной модуль, необходимый абсолютно для всех приложений Spring. Он предоставляет классы для использования другим модулям.

Spring-context — обеспечивает работу контекста IoC-контейнера.

Теперь у нас есть все, чтобы реализовать предыдущий пример, но уже с использованием Spring. В предыдущим коде создадим интерфейс **Camera** с единственным методом **doPhotograph()**, а наш предыдущий класс будет называться **CameraImpl** и реализовывать данный интерфейс. Делаем так, следуя правилу хорошего тона разработки, позволяющему в дальнейшем расширять приложение. Но основная причина связана с особенностями контекста Spring.

Клиентский код программы:

|  |
| --- |
| public class Client {    public static void main(String[] args) {      ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");      Camera camera = context.getBean("camera", Camera.class);      camera.doPhotograph();    }  } |

Теперь в роли класса **Assistant** выступает интерфейс **ApplicationContext**. Его метод **getBean** предоставляет клиенту настроенную камеру.

**Bean** — это объект любого класса, который управляется контейнером Spring (подробнее о bean — в следующем уроке).

Каковы особенности данного кода по сравнению с аналогичным, написанным без использования Spring? Для создания контекста, который играет роль нашего ассистента, необходимо передать параметры. В предыдущем примере мы ничего не передавали, так как точно знали, что класс **Assistant** нужен только для настройки и получения объекта фотоаппарата. А так как Spring является фреймворком, то и управлять он может абсолютно любыми объектами. Чтобы указывать Spring, какими именно объектами следует управлять (поместить в контекст) и какие зависимости необходимо удовлетворить, контексту передается определенная информация. В данном случае она находится во внешнем конфигурационном файле **config.xml**. Для чтения этой информации из конфигурационного файла (создания контекста) используется конкретная реализация **ApplicationContext** — **ClassPathXmlApplicationContext**. Она прекрасно «понимает» XML-язык.

**Конфигурирование Spring**

В упрощенном виде конфигурация Spring проходит в два этапа:

* инициализация контекста;
* создание, настройка, добавление компонентов (бинов) в контекст.

Виды конфигураций в Spring:

* XML-конфигурация;
* JavaConfig;
* использование аннотаций.

Можно сделать вывод, что Spring обладает двумя самостоятельными способами конфигураций. А третий (использование аннотаций) может применяться только на втором этапе совместно с одним из видов «самостоятельной» конфигурации. То есть можем применять:

* либо XML-конфигурацию + использование аннотаций,
* либо JavaConfig + использование аннотаций,
* или два «самостоятельных» способа конфигурации без аннотаций.

Рассмотрим каждый из этих способов.

**Конфигурирование с использованием XML**

Конфигурация через XML — первый вариант, который поддерживал Spring. Посмотрим в конфигурационный файл **config.xml**:

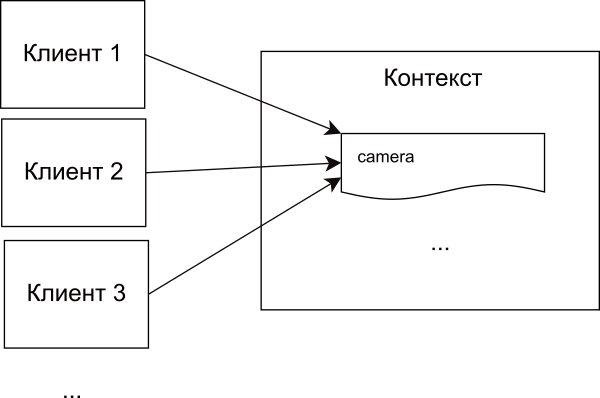
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"    xsi:schemaLocation="http://www.springframework.org/schema/beans    http://www.springframework.org/schema/beans/spring-beans.xsd">    <bean id="cameraRoll" class="ru.geekbrains.lesson1.ColorCameraRoll" />      <bean id="camera" class="ru.geekbrains.lesson1.CameraImpl">      <property name="cameraRoll">        <ref bean="cameraRoll" />      </property>    </bean>  </beans> |

Конфигурационный файл предназначен для указания контейнеру объектов, которыми он должен управлять (бинов). Основной схемой и пространством имен по умолчанию является <http://www.springframework.org/schema/beans/spring-beans.xsd> и <http://www.springframework.org/schema/beans>.

Тег **<bean … />** позволяет:

* создать объект определенного класса, который сразу же станет бином;
* внедрить объект другого класса в свойство данного бина, при этом данный объект тоже должен быть объявлен в конфигурационном файле.

Spring по умолчанию создает одиночные объекты, и все вызовы **getBean(...)** будут возвращать один и тот же созданный контейнером объект с указанным в скобках идентификатором. Схема доступа к данному объекту изображена на следующей схеме:



Чем это может быть полезно и как переопределить данное поведение — узнаем позже.

Теперь разберем каждую запись построчно:

|  |
| --- |
| <bean id="cameraRoll" class="ru.geekbrains.lesson1.ColorCameraRoll" /> |

С помощью тега **bean** объявляется объект класса **ColorCameraRoll**. Для него, как и в случае создания объекта в Java-коде через **new**, необходимо имя, с помощью которого можно обращаться к нему в дальнейшем. В XML-конфигурации в роли имени выступает значение атрибута ***id***. По этому идентификатору к бину можно обращаться как в файле конфигурации, так и получать его из контекста, передавая id в метод **getBean(...)**. Для указания класса создаваемого бина необходимо задать полное имя класса в атрибуте ***class***. Все просто, но есть особенность. Ранее мы оговаривали, что класс **ColorCameraRoll** реализует интерфейс **CameraRoll**. Соответственно, в Java-коде возможны два варианта создания данного объекта: либо через ссылку на сам класс ColorCameraRoll (1), либо через ссылку на интерфейс CameraRoll (2).

|  |
| --- |
| ColorCameraRoll cameraRoll = new ColorCameraRoll(); // (1) |

Или вот так:

|  |
| --- |
| CameraRoll cameraRoll = new ColorCameraRoll(); // (2) |

На самом деле отличия между этими двумя вариантами незначительны, но Spring реализует именно второй. И при извлечении бина из контекста с помощью метода **getBean(...)** в параметры данного метода нужно передавать интерфейс, а не конкретную реализацию. Сейчас это надо запомнить, а причины разберем позже.

Перейдем ко второму объявлению конфигурационного файла **config.xml**:

|  |
| --- |
| <bean id="camera" class="ru.geekbrains.lesson1.CameraImpl">    <property name="cameraRoll">      <ref bean="cameraRoll" />    </property>  </bean> |

В данном случае происходит создание объекта класса **CameraImp** c идентификатором **camera**. Чтобы Spring-контейнер самостоятельно вставил фотопленку в фотоаппарат, используем:

|  |
| --- |
| <property name="cameraRoll">    <ref bean="cameraRoll" />  </property> |

Тег **property** дает доступ к свойству нашего объекта класса **camera**. В атрибуте **name** указывается имя данного свойства, а так как пленку мы уже создали, то между двумя тегами **property** необходимо поместить ссылку (reference) на ранее созданный **bean**. Для этого и используется тег **ref** с атрибутом **bean**, в котором указывается id ранее созданного бина.

Все это эквивалентно Java-коду в предыдущем варианте, в котором Spring не использовался:

|  |
| --- |
| CameraRoll cameraRoll = new ColorCameraRoll();  Camera camera = new Camera();  camera.setCameraRoll(cameraRoll); |

Если некоторый бин необходим для одноразового внедрения и получать его из контекста не требуется, то XML-конфигурацию можно изменить следующим образом:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"    xsi:schemaLocation="http://www.springframework.org/schema/beans  <http://www.springframework.org/schema/beans/spring-beans.xsd>">    <bean id="camera" class="ru.geekbrains.lesson1.CameraImpl">      <property name="cameraRoll">        <bean  class="ru.geekbrains.lesson1.ColorCameraRoll" />      </property>    </bean>  </beans> |

Данный вариант эквивалентен следующему Java-коду:

|  |
| --- |
| Camera camera = new Camera();  camera.setCameraRoll(new ColorCameraRoll()); |

**Конфигурирование с использованием аннотаций**

С версии Spring 2.5 стало возможным убрать часть кода из XML-конфигурации с помощью специальных аннотаций. Например, чтобы в Spring-контейнере оказался объект фотопленки, достаточно над классом фотопленки указать аннотацию @Component:

|  |
| --- |
| @Component("cameraRoll")  public class ColorCameraRoll implements CameraRoll {    public void processing() {      System.out.println("-1 цветной кадр");    }  } |

Первая аннотация заставляет Spring-контейнер создать объект класса, к которому применена аннотация, с *id*, указанным в скобках. Данный объект сразу же будет являться бином — а значит, компонентом, управляемым контейнером.

Это позволит Spring-контейнеру создать объект данного класса, сделать его бином и поместить в контекст. Id бина указывается в аннотации в скобках, здесь это **cameraRoll**. По этому id объект будет доступен в контексте, и к нему можно обращаться в XML-конфигурации. Но как Spring увидит данную аннотацию? Стоит отметить, что клиентский код при таком подходе не изменится: нам все так же придется создавать контекст с помощью внешнего конфигурационного XML-файла. Но благодаря этой аннотации можно избавиться от тегов **<bean…>** в файле **config.xml**. В нем же необходимо указать Spring-контейнеру, где искать классы, помеченные данной аннотацией.

Сделать это можно следующим образом:

1. В **config.xml** подключаем пространство имен (<http://www.springframework.org/schema/context/spring-context.xsd>) и схему (<http://www.springframework.org/schema/context>);
2. Чтобы Spring-контейнер «знал», где искать объявленные с помощью аннотаций компоненты, в config-файл добавляем строку:

|  |
| --- |
| <context:component-scan base-package="ru.geekbrains.lesson1" /> |

Теперь объект класса **CameraRoll** создается и помещается в контекст с помощью аннотаций. Конфигурационный файл приобрел следующий вид:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"    xmlns:context="http://www.springframework.org/schema/context"    xsi:schemaLocation="http://www.springframework.org/schema/beans    http://www.springframework.org/schema/beans/spring-beans.xsd    http://www.springframework.org/schema/context  <http://www.springframework.org/schema/context/spring-context.xsd>">    <context:component-scan base-package="ru.geekbrains.lesson1" />    <bean id="camera" class="ru.geekbrains.CameraImpl">      <property name="cameraRoll">        <ref bean="cameraRoll" />      </property>    </bean>  </beans> |

**JavaConfig**

Создание контекста Spring может происходить вообще без использования XML-конфигурации. Всю конфигурацию можно вынести в отдельный Java-класс, помеченный аннотацией @Configuration:

|  |
| --- |
| @Configuration  public class AppConfig {    @Bean(name="cameraRoll")    public CameraRoll cameraRoll() {      return new ColorCameraRoll() ;    }    @Bean(name="camera")    public Camera camera(CameraRoll cameraRoll){      Camera camera = new CameraImpl();      camera.setCameraRoll(cameraRoll);      return camera;    }  } |

В данном листинге присутствуют две новые аннотации:

* **@Configuration** — аннотация, указывающая на то, что данный Java-класс является классом конфигурации;
* **@Bean** — используется для аннотирования методов, создающих бины в классе, помеченном аннотацией @Configuration. Аналог тега <bean…../> в XML-конфигурации.

Следующий листинг этого Java-кода:

|  |
| --- |
| @Bean(name="cameraRoll")  public CameraRoll cameraRoll() {    return new ColorCameraRoll() ;  }    @Bean(name="camera")  public Camera camera(CameraRoll cameraRoll){    Camera camera = new CameraImpl();    camera.setCameraRoll(cameraRoll);    return camera;  } |

… можно представить как:

|  |
| --- |
| <bean id="cameraRoll" class="ru.geekbrains.lesson1.ColorCameraRoll" />  <bean id="camera" class="ru.geekbrains.lesson1.CameraImpl">    <property name="cameraRoll">      <ref bean="cameraRoll" />    </property>  </bean> |

Клиентский код:

|  |
| --- |
| ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);  Camera camera = context.getBean("camera",Camera.class);  camera.doPhotograph(); |

Единственным изменением стало то, что теперь для создания контекста применяется другая реализация **ApplicationContext**, которая имеет имя **AnnotationConfigApplicationContext**. В параметр конструктора передается класс, который был описан ранее.

Совместно применять JavaConfig и аннотации можно, добавляя к классу конфигурации аннотации ***@ComponentScan(...)****.* В нашем случае это будет выглядеть следующим образом:

|  |
| --- |
| @Configuration  @ComponentScan("ru.geekbrains.lesson1")  public class AppConfig {    @Bean(name="camera")    public Camera camera(CameraRoll cameraRoll){      Camera camera = new CameraImpl();      camera.setCameraRoll(cameraRoll);      return camera;    }  } |

Это будет полностью эквивалентно следующему XML:

|  |
| --- |
| <context:component-scan base-package="ru.geekbrains.lesson1" />  <bean id="camera" class="ru.geekbrains.lesson1.CameraImpl">    <property name="cameraRoll">      <ref bean="cameraRoll" />    </property>  </bean> |

**Советы в выборе способа конфигурации**

XML-конфигурация и JavaConfig являются двумя равнозначными способами конфигурации в Spring. Но JavaConfig все же обладает весомыми преимуществами перед XML-конфигурацией:

1. Используется Java-код, а значит, не нужно заботиться о xsd-схемах и XML-тегах.
2. Можно выявить ошибки на этапе написания конфигурационного класса.
3. Больше гибкости за счет работы с объектами и методами.

Вывод: наилучшим способом конфигурации является JavaConfig, но по возможности на протяжении курса будет приводиться и XML-аналог конфигурации.

Стоит ли применять конфигурацию с помощью аннотаций совместно с JavaConfig? Однозначно стоит. Оптимальный способ — условие, согласно которому все бины, необходимые для инфраструктуры приложения (источники данных, менеджеры транзакций), объявляются непосредственно в классе JavaConfig путем создания метода и применения к нему аннотации *@Bean*. А ко всем классам, реализующим написанную нами бизнес-логику (либо к классам, предназначенным для хранения какой-либо информации), применяются специальные аннотации (*@Component, @Service, @Repository* и т. п.). Рассмотрим их в следующем уроке.

**Внедрение зависимостей в Spring**

Вернемся к теории и вспомним виды Dependency Injection:

* внедрение через конструктор;
* внедрение через сеттер;
* внедрение на уровне поля.

Почти во всех случаях предпочтительным является внедрение через сеттер. Именно так мы и делали, но выполняли это непосредственно в классе конфигурации (в случае JavaConfig) либо в XML-файле (в случае XML-конфигурации). Попрактикуемся в возможностях, которые поддерживает Spring для внедрения зависимостей.

Что будем внедрять:

* значения примитивных типов и строк;
* бинов.

Как будем внедрять?

* с помощью специальных тегов в XML-конфигурации;
* с помощью методов в JavaConfig;
* непосредственно в коде, используя аннотацию @Autowired.

**Внедрение значений примитивных типов и строк**

Представим, что у нас есть интерфейс **HelloMan**, который объявляет единственный метод приветствия:

|  |
| --- |
| public interface HelloMan {    public void helloSay();    public String getName();    public void setName(String name);  } |

И его реализация **HelloManOnceSay**:

|  |
| --- |
| public class HelloManOnceSay implements HelloMan {    private String name;    public HelloManOnceSay(){    }    public HelloManOnceSay(String name){      this.name = name;    }    public String getName() {      return name;    }    public void setName(String name) {      this.name = name;    }    public void helloSay(){      System.out.println("Hello," + this.name);    }  } |

Для работы метода **helloSay()** нам необходимо обеспечить зависимость класса в поле **name**, которая имеет тип **String**.

Внедрить зависимости при использовании XML-конфигурации мы можем несколькими путями: через конструктор и через сеттер. Если применять только XML-конфигурацию, объявление бина и обеспечение внедрения зависимости типа String *через конструктор* будет выглядеть следующим образом:

|  |
| --- |
| <bean id="helloMan" class="ru.geekbrains.lesson1.HelloManOnceSay">    <constructor-arg value="Yuri" />  </bean> |

Для тега **constructor-arg** возможно определение двух дополнительных атрибутов: ***type*** и ***index***. Они указывают на тип внедряемого значения и его порядковый номер в конструкторе класса.

|  |
| --- |
| <bean id="helloMan" class="ru.geekbrains.lesson1.HelloManOnceSay">    <constructor-arg type="java.lang.String" index="0" value="Yuri" />  </bean> |

Внедрение *через сеттер* (предпочтительный способ):

|  |
| --- |
| <bean id="helloMan" class="net.zt.funcode.lesson1.HelloManOnceSay">    <property name="name" value="Yuri" />  </bean> |

При использовании JavaConfig внедрение *через конструктор* выглядит довольно просто:

|  |
| --- |
| @Bean(name="helloMan")  public HelloMan helloMan(@Value("Yuri") String name) {    HelloMan helloMan = new HelloManOnceSay(name);    return helloMan;  } |

Аннотация **@Value** внедряет примитивное значение в элемент, к которому применена данная аннотация. Для внедрения зависимости **name** через сеттер достаточно воспользоваться пустым конструктором и использовать метод **setName(name)**.

Но ранее мы упоминали, что бины классов, реализующих бизнес-логику, объявляются вне конфигурации с помощью аннотации **@Component** и подобных. Но как быть с зависимостями данного бина? Применяем ту же аннотацию **@Value**:

|  |
| --- |
| @Component  public class HelloManOnceSay implements HelloMan {    @Value("Yuri")    private String name;    public HelloManOnceSay(){    }    public HelloManOnceSay(String name){      this.name = name;    }    public String getName() {      return name;    }      public void setName(String name) {      this.name = name;    }    public void helloSay() {      System.out.println("Hello, " + this.name);    }  } |

Кроме того, аннотацию **@Value** можно применить и к параметру конструктора, и к сеттеру **setName(...)**. Внедрение числовых типов происходит аналогичным образом.

**Внедрение бинов**

Для внедрения бинов применяется такой же подход. Но в случае применения XML-конфигурации и *внедрения через конструктор* вместо такого тега:

|  |
| --- |
| <constructor-arg value="..." /> |

… используется следующая конструкция:

|  |
| --- |
| <constructor-arg ref="..." /> |

Здесь значением **ref** будет имя (id, идентификатор) нашего бина.

Для внедрения *через сеттер* применяется следующая конструкция:

|  |
| --- |
| <property name="cameraRoll">    <ref bean="cameraRoll" />  </property> |

Для внедрения в JavaConfig применяется конструкция, которую мы уже рассматривали:

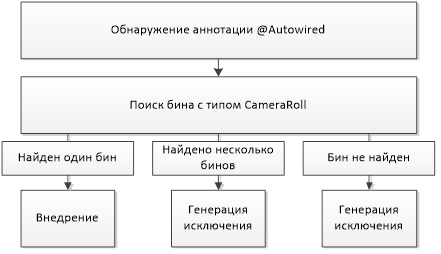
|  |
| --- |
| @Bean(name="cameraRoll")  public CameraRoll cameraRoll() {    return new ColorCameraRoll() ;  }    @Bean(name="camera")  public Camera camera(CameraRoll cameraRoll) {    Camera camera = new CameraImpl();    camera.setCameraRoll(cameraRoll);    return camera;  } |

В этом листинге происходит внедрение через сеттер. Чтобы сделать подобное через конструктор, необходимо передать параметр **cameraRoll** в параметры оператора **new** (если класс имеет конструктор с данной сигнатурой). Самое интересное нас ждет при внедрении объектов через аннотации. Необходимо использовать аннотацию **@Autowired**. Перепишем класс камеры, применяя ее:

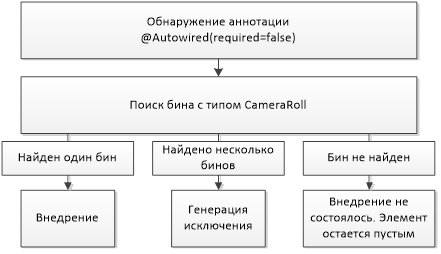
|  |
| --- |
| @Component("camera")  public class CameraImpl implements Camera {    private CameraRoll cameraRoll;      public CameraRoll getCameraRoll() {      return cameraRoll;    }    @Autowired    public void setCameraRoll(CameraRoll cameraRoll) {      this.cameraRoll = cameraRoll;    }    public void doPhotograph() {      System.out.println("Сделана фотография!");      cameraRoll.processing();    }  } |

Эта аннотация может применяться к полю, сеттеру и конструктору.

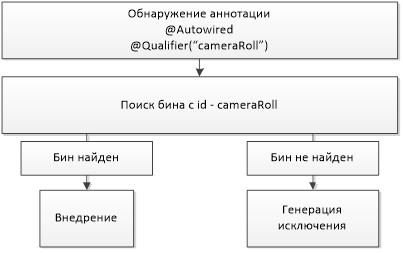
В данной программе алгоритм внедрения будет работать следующим образом:



Из схемы видно, что поиск бина, который станет кандидатом на внедрение, осуществляется по типу. Если в контексте есть несколько бинов заданного типа или бин не будет найден, то генерируется исключение **NoSuchBeanDefinitionException**. В обоих случаях исключения можно избежать. Например, если атрибут **required** аннотации **@Autowired** установить в значение **false**: **@Autowired(required=false)**. Тогда в случае, если нужный бин отсутствует, в элемент с данной аннотацией значение внедрено не будет.



Чтобы избежать исключения, когда в контексте имеется несколько бинов данного типа, необходимо использовать аннотацию **@Qualifier(“beanName”)**. Она позволяет указать id бина, который должен быть внедрен.



**Область видимости бинов**

Что будет, если клиент случайно сломает фотоаппарат? Времени на починку нет – и он попросит другую камеру.

Модифицируем наш код. Интерфейс фотоаппарата примет вид:

|  |
| --- |
| **public** **interface** Camera {    CameraRoll getCameraRoll();   **void** setCameraRoll(CameraRoll cameraRoll);   **void** doPhotograph();   **void** breaking();   **boolean** isBroken(); } |

Здесь вызов метода **breaking()** будет имитировать поломку фотоаппарата. А с помощью метода **isBroken()** сможем проверить, является ли фотоаппарат работоспособным. Реализация интерфейса будет выглядеть следующим образом:

|  |
| --- |
| @Component(**"camera"**) **public** **class** CameraImpl **implements** Camera {    @Autowired   @Qualifier(**"cameraRoll"**)   **private** CameraRoll cameraRoll;     @Value(**"false"**)   **private** **boolean** broken;      **public** CameraRoll getCameraRoll() {     **return** cameraRoll;   }    **public** **void** setCameraRoll(CameraRoll cameraRoll) {     **this**.cameraRoll = cameraRoll;   }    **public** **boolean** isBroken() {     **return** broken;   }    **public** **void** breaking(){     **this**.broken=**true**;   }    **public** **void** doPhotograph() {      **if** (isBroken()) {       System.out.println(**"Фотоаппарат сломан!"**);       **return**;     }     System.out.println(**"Сделана фотография!"**);     cameraRoll.processing();    }  } |

Мы объявили дополнительное поле **broken**, которое будет иметь логическое значение, позволяющее проверить работоспособность фотоаппарата. Также объявлены два метода: **isBroken()** (проверить, является ли фотоаппарат сломанным) и **breaking()**(сломать фотоаппарат). Кроме того, модифицирован метод **doPhotograph()**,в котором проверяется, работоспособна ли камера.

Смоделируем ситуацию: клиент получает фотоаппарат у своего помощника и случайно его ломает. Сделать фото не получается – он просит еще один фотоаппарат и пытается сделать фото. На языке кода это будет выглядеть так:

|  |
| --- |
| **public** **class** Client {   **public** **static** **void** main(String[] args) {     ApplicationContext context = **new** AnnotationConfigApplicationContext(AppConfig.class);  *// Получает фотоаппарат*     Camera camera = context.getBean(**"camera"**, Camera.class);      *// Ломает фотоаппарат*     camera.breaking();     *// Пытается сделать фото. Неудача!*     camera.doPhotograph();      *// Просит еще один фотоаппарат*     camera = context.getBean(**"camera"**, Camera.class);     *// Пытается сделать фото*     camera.doPhotograph();   } } |

Происходит странное: второй фотоаппарат тоже сломан. Причина в том, что это был один и тот же фотоаппарат. И каждый раз, когда клиент будет просить камеру, он будет получать прежнюю.

Помощник в данной ситуации не так надежен, как казалось.  Дело в том, что фотоаппарат является объектом «одиночка» (Singleton), и все обращения происходят к одному и тому же объекту.

По умолчанию все компоненты Spring являются синглтонами, и в большинстве ситуаций такой подход оптимален, но не в нашем случае. Чтобы переопределить данное поведение, необходимо добавить в класс фотоаппарата следующий код:

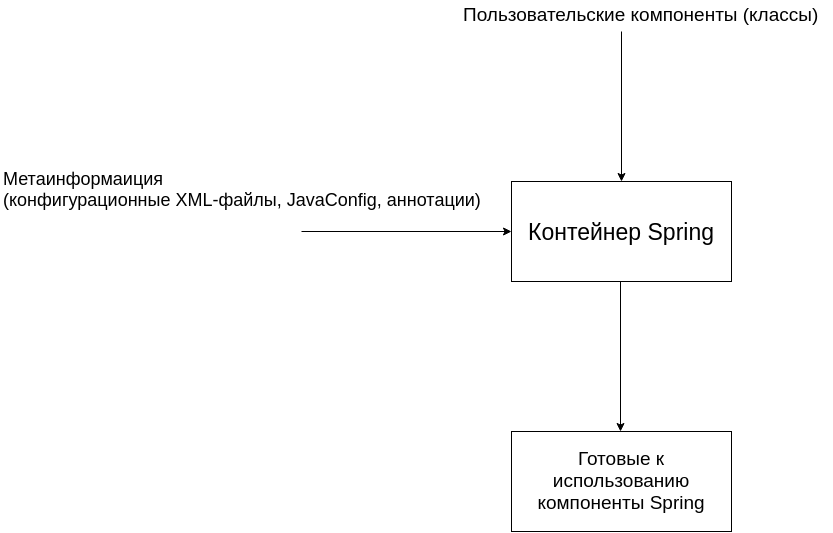
|  |
| --- |
| @Component(**"camera"**) @Scope(**"prototype"**) **public** **class** CameraImpl **implements** Camera {     *// ...* } |

Мы добавили аннотацию **@Scope(“prototype”)**. Она говорит Spring, что при каждом **getBean** (**camera, Camera.class**) необходимо возвращать новый объект фотоаппарата.

Все бины, создаваемые в **JavaConfig** с помощью аннотации **@Bean**, тоже являются синглтонами (аналогично и в XML). Это поведение можно изменить в обоих видах конфигурационных файлов. Подробнее об этом читайте в дополнительном материале к уроку.

**Этапы инициализации контекста**

Прежде, чем использовать богатый функционал Spring, компонентам необходимо попасть в контейнер в фреймворке. Если смотреть глобально, то это выглядит следующим образом:



В итоге все бины оказываются в контексте, который является объектом класса **ApplicationContext**. Но прежде чем туда попасть, они проходят немалый путь, на определенных этапах которого мы можем внести свой вклад в создание бина. Весь процесс создания и инициализации бина делится на четыре этапа:

**Этап 1**. Производится чтение данных из конфигурационного файла. На основании полученных данных создаются специальные объекты BeanDefinition, которые несут в себе информацию о бинах, которые надо создать. Эти объекты попадают в BeanFactory.

**Этап 2**. Производится настройка объектов класса BeanDefinition.

**Этап 3**. Производится конечная настройка созданных бинов (в том числе, внедрение зависимостей)

**Этап 4**. BeanFactory создает бины, используя информацию, хранящуюся в объектах BeanDefinition.

В данной схеме присутствуют два этапа настройки (Этап 2 и Этап 4). Именно на этих этапах идет обработка большей части аннотаций Spring.

Этап 1

На первом этапе в зависимости от вида конфигурации используются различные классы, реализующие интерфейс **ApplicationContext**, при создании которых, в конструктор передается ссылка на файл конфигурации(это может быть xml файл, или Java класс). После подготовки объекта типа ApplicationContext, формируются объекты типа **BeanDefinition**, содержащие информацию об объектах, которые необходимо будет создать. Все **BeanDefinition** хранятся в определенном контейнере – **BeanFactory**. Объект **BeanDefinition** содержит следующую информацию:

* **Class** – полный путь к классу, объектом которого будет являться будущий бин;
* **Scope** – область видимости (по умолчанию **singleton**);
* **Abstract** – указывает, является ли бин абстрактным. Такой бин может быть объектом абстрактного класса, но в случае с XML-конфигурацией класс можно не указывать – достаточно задать лишь свойства. Бины, являющиеся абстрактными, используются только для создания дочерних бинов;
* и другие.

Кроме **BeanDefinition**, описывающих пользовательские компоненты, в этом контейнере будут содержаться и **BeanDefinition**, описывающие служебные бины Spring. Именно на данном этапе Spring обрабатывает уже известные нам аннотации: **@Configuration**, **@ComponentScan**, **@Component**, **@Bean** и другие. На этом этапе вмешиваться в работу Spring не стоит. Итогом этапа является определенное количество созданных объектов **BeanDefinition**, содержащих информацию о том, какие объекты и с какими параметрами необходимо создать.

Этап 2

На втором этапе происходит настройка созданных **BeanDefinition**. Обрабатываются только те **BeanDefinition**, классы которых помечены определенными аннотациями, или аннотации, имеющие определенные параметры – например, **@Value(“${property.password}”** (подробнее о том, что означает данный параметр, мы поговорим на следующих уроках).

Каким образом происходит эта настройка? Что необходимо сделать, чтобы получить доступ ко всем **BeanDefinition**? Нужен доступ к контейнеру, в котором они хранятся – **BeanFactory**. Для настройки потребуется класс-настройщик, написанный нами. Но каким образом в него попадет **BeanFactory**? Нам не придется выискивать данный контейнер – Spring сам предоставит **BeanFactory** классу-настройщику. Для этого необходимо выполнить два условия:

* Класс-настройщик должен быть компонентом Spring (т.е. иметь аннотацию **@Component**, а путь к данному классу должен содержаться в конфигурационном файле или в классе нашего приложения);
* Реализовывать интерфейс **BeanFactoryPostProcessor** и его единственный метод **postProcessBeanFactory**, который и принимает в качестве параметра **BeanFactory**.

Ниже приведен пример реализации собственного класса-настройщика, который выводит информацию о всех **BeanDefinition**, содержащихся в BeanFactory:

|  |
| --- |
| @Component **public** **class** TestBeanFactoryPostProc **implements** BeanFactoryPostProcessor {   **public** **void** postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) **throws** BeansException {      *// Получение имен  BeanDefinition всех бинов, объявленных пользователем*     String[] beanDefinitionNames = beanFactory.getBeanDefinitionNames();      *// Перебор массива для получения доступа к каждому имени*     **for**(String name: beanDefinitionNames) {          *// Получение BeanDefinition по имени*       BeanDefinition beanDefinition = beanFactory.getBeanDefinition(name);       *// Вывод информации о BeanDefinition*       System.out.println(beanDefinition.toString());     }   } } |

При запуске можно увидеть, что объектов **BeanDefinition** на самом деле больше, чем ожидалось. Это связано с тем, что Spring использует собственные служебные бины для настройки компонентов, помеченных определенными аннотациями (например, **BeanDefinition**, описывающего бин класса **TestBeanFactoryPostProc**, который приведен выше).

*Создание классов, реализующих* ***BeanFactoryPostProcessor****, позволит повлиять на то, каким будет бин еще до его создания.*

Вернемся к примеру с фотоаппаратом – чтобы понять, как Spring на данном этапе обрабатывает некоторые из своих аннотаций, а также ощутить всю мощь **BeanFactoryPostProcessor**.

Представим ситуацию: клиент в своих требованиях указывает помощнику, что хочет фотографировать на черно-белую пленку. Но она уже давно не производится – вместо нее все применяют цветную. Помощник это понимает и покупать будет именно цветную пленку, тем самым вмешиваясь в требования клиента.

Отредактируем наш код. Раз клиент требует черно-белую пленку, то аннотацию **@Component** стоит **удалить** из класса **ColorCameraRoll** и **добавить** в класс **BlackAndWhiteCameraRoll**. Ведь клиенту цветная пленка совсем не интересна – значит, не обязательно класс **ColorCameraRoll** объявлять компонентом Spring:

|  |
| --- |
| @Component(**"cameraRoll"**) **public** **class** BlackAndWhiteCameraRoll **implements** CameraRoll {   **public** **void** processing(){     System.out.println(**"-1 черно-белый кадр"**);   } } |

Не каждый помощник будет знать о том, что черно-белая пленка уже давно не производится. Значит, класс этой фотопленки должен быть как-то помечен. Единственным верным способом будет создание аннотации. Она должна недвусмысленно говорить помощнику о том, что этот вид фотопленки больше не производится и нужно искать другую.

|  |
| --- |
| @Retention(RetentionPolicy.RUNTIME) @Target(ElementType.TYPE) **public** @interface UnproducableCameraRoll {   Class usingCameraRollClass(); } |

Применение данной аннотации означает, что фотопленка этого класса больше не производится. Значение поля **usingCameraRollClass** должно указывать на класс пленки, которую следует использовать вместо данной.

Применим  эту аннотацию к классу черно-белой фотопленки:

|  |
| --- |
| @Component(**"cameraRoll"**) @UnproducableCameraRoll(usingCameraRollClass=ColorCameraRoll.class) **public** **class** BlackAndWhiteCameraRoll **implements** CameraRoll {   **public** **void** processing(){     System.out.println(**"-1 черно-белый кадр"**);   } } |

Значением поля **usingCameraRollClass** будет  класс цветной фотопленки.

Теперь необходимо написать класс-настройщик, который будет обнаруживать данную аннотацию и изменять класс фотопленки, указанный в **BeanDefinition**, на класс, указанный в параметре аннотации **@UnproducableCameraRoll**. С учетом тех требований, которые выдвигаются к подобному классу, он будет иметь следующий вид:

|  |
| --- |
| @Component **public** **class** UnproducableCameraRollBeanFactoryPostProcessor **implements** BeanFactoryPostProcessor {    **public** **void** postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) **throws** BeansException {      *// Получаем имена всех BeanDefinition для доступа к каждому из них*     String[] beanDefinitionNames = beanFactory.getBeanDefinitionNames();     *// Перебираем все имена*     **for**(String name: beanDefinitionNames){       *// Получаем BeanDefinition по имени*       BeanDefinition beanDefinition = beanFactory.getBeanDefinition(name);         */\*Получаем имя класса создаваемого бина, чтобы проверить,        \* содержит ли он аннотацию UnproducableCameraRoll        \*/*        String className = beanDefinition.getBeanClassName();         **try** {         *// Получаем класс по имени*         Class<?> beanClass = Class.forName(className);           */\*Пытаемся получить объект аннотации и ее значение,         \* если  класс не содержит данную аннотацию, то  метод вернет null         \*/*         UnproducableCameraRoll annotation = (UnproducableCameraRoll)beanClass.getAnnotation(UnproducableCameraRoll.class);          *// Проверяем, содержал ли класс эту аннотацию*         **if**(annotation != **null**) {           *// Получаем значение, указанное в параметрах аннотации (класс пленки, которую необходимо использовать)*           Class usingCameraRollName = annotation.usingCameraRollClass();  *// Меняем класс будущего бина*           beanDefinition.setBeanClassName(usingCameraRollName.getName());          }       } **catch** (ClassNotFoundException e) {         e.printStackTrace();       }     }   } } |

Запустив клиентский код, вы сможете убедиться, что помощник действительно подменил класс фотопленки, вопреки требованиям клиента.

Для реализации подобных трюков есть одно очень важное условие. Еще раз взглянем на объявление атрибута, в который «вживляется» фотопленка:

|  |
| --- |
| @Component(**"camera"**) **public** **class** CameraImpl **implements** Camera {    @Autowired   @Qualifier(**"cameraRoll"**)   **private** CameraRoll cameraRoll;   *// ...* } |

Данный атрибут имеет тип интерфейса **CameraRoll**. Его реализуют оба класса фотопленки – значит, независимо от того, какую фотопленку в итоге придется вставить в фотоаппарат, внедрение пройдет успешно.

Теперь вспомним строку из нашего клиентского кода:

|  |
| --- |
| Camera camera = context.getBean("camera", Camera.class); |

Бин **camera** класса **CameraImpl** извлекается из контекста по интерфейсу, который реализует данный класс. Это помогает нам не получить исключение в случае, если помощник подменит класс фотокамеры другим классом, который реализует данный интерфейс.

Этап 3

На этом этапе **BeanFactory**, используя **BeanDefinition**, создает бины. Данный процесс является внутренним процессом Spring, и нет смысла влиять на него. Важно отметить, что на этой стадии бины лишь создаются, но их зависимости еще не удовлетворены.

Этап 4

На четвертом этапе мы имеем созданные бины, но зависимости еще не внедрены, в том числе и простые значения, внедряемые аннотацией **@Value**. Значит, бины еще не готовы к использованию. Чтобы внедрить зависимости, Spring использует классы-настройщики, которые и обрабатывают аннотации **@Value**, **@Autowired** и другие. Мы без проблем можем реализовать собственный класс-настройщик, но для этого он должен выполнять два условия:

* Класс-настройщик должен являться компонентом Spring (иметь аннотацию **@Component**, а путь к данному классу должен содержаться в конфигурационном файле или классе нашего приложения);
* Реализовывать интерфейс **BeanPostProcessor** и его методы:

|  |
| --- |
| **public** **interface** BeanPostProcessor {   Object postProcessBeforeInitialization(Object bean, String beanName) **throws** BeansException;   Object postProcessAfterInitialization(Object bean, String beanName) **throws** BeansException; } |

В отличие от класса, реализующего интерфейс **BeanFactoryPostProcessor**, реализация этого интерфейса позволит получать доступ к каждому бину поочередно. В каждый из методов передается сам бин и его имя.  Данные методы имеют следующее описание:

* **postProcessBeforeInitialization** – метод, вызываемый до инициализации бина (термин «инициализация» в данном контексте довольно относителен: для Spring это означает вызов пользовательского init-метода, о котором будет рассказано далее). На данном этапе бин создан, и в него уже внедрены зависимости, которые помечены аннотациями Spring (**@Autowired**, **@Value** и т.п.). Классы-настройщики Spring всегда будут вызываться раньше, чем реализованные пользователем;
* **postProcessAfterInitialization** – метод, который выполняется после инициализации (после вызова init-метода). После него настроенные бины попадают непосредственно в контейнер бинов.

Предположим, что помощник перед покупкой фотоаппарата решил проверить его работоспособность. Для этого ему необходимо самому попробовать сделать фотографию. Для осуществления этой затеи реализуем собственный **BeanPostProcessor**:

|  |
| --- |
| @Component  public class PhotocameraTestBeanPostProcessor implements BeanPostProcessor {    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {      // В данном методе просто возвращаем бин      return bean;    }    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {      // Находим бин класса фотокамеры      if (bean instanceof Camera) {        System.out.println("Делаю пробное фото!");  // Делаем пробное фото  ((Camera) bean).doPhotograph();  System.out.println("Отлично! Работает!");      }      return bean;    }  } |

По завершении этапа бины полностью готовы к использованию.

**Жизненный цикл бина**

Все, что было рассмотрено ранее, относится к этапам инициализации контекста. Но сам бин об этих этапах ничего не знает. Бины Spring обладают жизненным циклом. Фактически, это дает возможность вызывать собственные методы бина на его жизненных этапах. Чтобы подобное стало возможным, необходимо как-то пометить метод и время его вызова. На практике используется несколько подходов.

Первый подход использует аннотации:

* **@PostConstruct** – метод инициализации, вызываемый после создания объекта и внедрения зависимостей (т.е. между методами **postProcessBeforeInitialization** и **postProcessAfterInitialization** интерфейса **BeanPostProcessor**);
* **@PreDestroy** – метод, вызываемый перед уничтожением бина.

Второй подход использует XML-атрибуты тега **<bean>**:

* **init-method**;
* **destroy-method**.

Оба подхода являются полными аналогами друг друга. Но если в вашем приложении используются оба подхода для одного и того же бина, то первыми будут вызываться методы, помеченные аннотацией.

Применение этих аннотаций возможно благодаря поддержке Spring стандарта JSR-250. Соответственно, для использования **@PostConstruct** и **@PreDestroy** необходимо подключать дополнительную зависимость.

В примере с фотокамерой реализуем метод инициализации, который выводит уведомление о том, что фотоаппарат готов к использованию.

Чтобы использовать аннотации жизненного цикла (в частности, **@PostConstruct**), необходимо добавить в файл **pom.xm**l следующую зависимость:

|  |
| --- |
| <dependency>      <groupId>javax.annotation</groupId>      <artifactId>jsr250-api</artifactId>      <version>1.0</version>  </dependency> |

В интерфейсе **Camera** объявим новый метод:

|  |
| --- |
| public void ready(); |

Реализуем этот метод в классе **CameraImpl** и пометим его аннотацией:

|  |
| --- |
| @Component("camera")  public class CameraImpl implements Camera {    @Autowired    private CameraRoll cameraRoll;      @Value("false")    private boolean broken;      public CameraRoll getCameraRoll() {      return cameraRoll;    }    public void setCameraRoll(CameraRoll cameraRoll) {      this.cameraRoll = cameraRoll;    }    public boolean isBroken() {      return broken;    }    public void breaking(){      this.broken = true;    }    public void doPhotograph(){      if (isBroken()) {        System.out.println("Фотоаппарат сломан!");  return;      }      System.out.println("Сделана фотография!");      cameraRoll.processing();    }      @PostConstruct    public void ready() {      System.out.println("Фотоаппарат готов к использованию!");    }  } |

При запуске клиентского кода получим оповещение о готовности к использованию. Данный метод будет вызван между двумя методами интерфейса **BeanPostProcessor**.

**Практическое задание**

1. Есть класс Product (id, название, цена). Товары хранятся в бине ProductRepository, в виде List<Product>, при старте в него нужно добавить 5 любых товаров. ProductRepository позволяет получить весь список или один товар по id. Создаем бин Cart, в который можно добавлять и удалять товары по id. Написать консольное приложение, позволяющее управлять корзиной. При каждом запросе корзины из контекста, должна создаваться новая корзина.
2. [ На удаление ] Попробовать настроить проект с использованием Spring и прописать различные настройки бинов. В качестве практического задания можете сдавать тестовый проект либо вопросы, которые возникли при изучении данной темы.

**Дополнительные материалы**

1. [Singleton и Prototype](http://docs.spring.io/spring/docs/3.0.0.M3/reference/html/ch04s04.html).
2. [Beans и Factory method](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html).
3. Крис Шефер, Кларенс Хо, Роб Харроп. Spring 4 для профессионалов.

**Используемая литература**

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Крис Шефер, Кларенс Хо, Роб Харроп. Spring 4 для профессионалов (4-е издание);
2. Крейг Уоллс. Spring в действии.

**Spring MVC**

MVC. Spring MVC. Контроллеры. Работа с формами. Представления. Конфигурирование Spring MVC. Контекст Spring MVC

**Оглавление**

[Немного теории](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.3znysh7)

[Обзор Spring MVC](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.q1zhzi3n0xjz)

[Обработка запросов в Spring MVC](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.2et92p0)

[Контроллеры](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.tyjcwt)

[Работа с формами](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.3dy6vkm)

[Контекст Spring MVC](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.1t3h5sf)

[Очень коротко об обработке запросов](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.i7lts9bppqlc)

[Практика](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.4d34og8)

[Создание проекта](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.2s8eyo1)

[Настройка проекта](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.d2mfvajpp93p)

[Конфигурирование](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.17dp8vu)

[Создание представления](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.3rdcrjn)

[Создание контроллера](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.26in1rg)

[Установка Apache Tomcat 9](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.1o8qxdaj6mj6)

[Запуск приложения из IntelliJ IDEA с помощью Maven](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.7qxfr1hw531c)

[Практическое задание](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.lnxbz9)

[Дополнительные материалы](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.35nkun2)

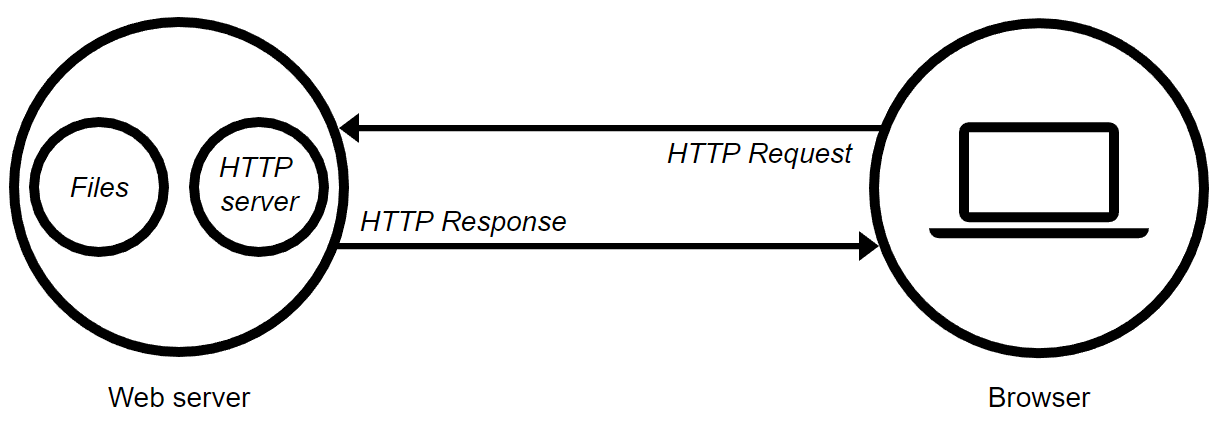
[Используемая литература](https://docs.google.com/document/d/14mU1CRobcPZPSkvgSFHdU0Hc9rUXtj8nsnVd-Isulm8/edit#heading=h.1ksv4uv)

**Немного теории**

Для начала давайте посмотрим что такое веб-сервер и сервер приложения. Понятие «веб-сервер» может относиться как к аппаратной начинке, так и к программному обеспечению. Или даже к обеим частям, работающим совместно.

1. С точки зрения "железа", «веб-сервер» — это компьютер, который хранит файлы сайта (HTML-документы, CSS-стили, JavaScript-файлы, картинки и другие) и доставляет их на устройство конечного пользователя (веб-браузер и т.д.). Он подключен к сети Интернет и может быть доступен через доменное имя, подобное mozilla.org.
2. С точки зрения ПО, веб-сервер включает в себя несколько компонентов, которые контролируют доступ веб-пользователей к размещенным на сервере файлам, как минимум — это *HTTP-сервер*. HTTP-сервер — это часть ПО, которая понимает [URL’ы](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/URL) (веб-адреса) и [HTTP](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/HTTP) (протокол, который ваш браузер использует для просмотра веб-страниц).

На самом базовом уровне, когда браузеру нужен файл, размещенный на веб-сервере, браузер запрашивает его через HTTP-протокол. Когда запрос достигает нужного веб-сервера ("железо"), сервер HTTP (ПО) принимает запрос, находит запрашиваемый документ (если нет, то сообщает об ошибке [404](https://developer.mozilla.org/ru/docs/Web/HTTP/Status/404)) и отправляет обратно, также через HTTP.



Чтобы опубликовать веб-сайт, необходим либо статический, либо динамический веб-сервер.

**Статический веб-сервер**, или стек, состоит из компьютера ("железо") с сервером HTTP (ПО). Мы называем это «статикой», потому что сервер посылает размещенные файлы в браузер «как есть».

**Динамический веб-сервер** состоит из статического веб-сервера и дополнительного программного обеспечения, чаще всего ***сервера приложения***и *базы данных*. Мы называем его «динамическим», потому что сервер приложений изменяет исходные файлы перед отправкой в ваш браузер по HTTP.

Например, для получения итоговой страницы, которую вы просматриваете в браузере, сервер приложений может заполнить HTML-шаблон данными из базы данных. Такие сайты, как MDN или Википедия, состоят из тысяч веб-страниц, но они не являются реальными HTML документами — лишь несколько HTML-шаблонов и гигантские базы данных. Эта структура упрощает и ускоряет сопровождение веб-приложений и доставку контента.

Теперь посмотрим на самый простой сценарий работы в веб. Допустим клиент с помощью любого браузера отправил HTTP запрос. Первым делом этот запрос получает веб-сервер (сервер приложения ни в коем случае не заменяет собой веб-сервер). Все что умеет делать веб-сервер это управлять файлами. Получая запрос от клиента веб-сервер сопоставляет запрос с файлом в файловой системе и затем отправляет файл обратно клиенту. Таким образом можно работать с различными файлами: HTML, PDF, ZIP, 7Z и др.

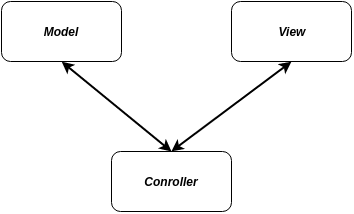
Если же попросить веб-сервер выполнить 1 + 1, то он не сможет этого сделать. Для выполнения на стороне сервера какой-либо логики необходимо передать входящий запрос нашему серверу приложения.

Когда клиент делает запрос, этот запрос получает веб-сервер.Веб-сервер считывает xml-файл конфигурации, предоставленный нашим сервером приложения, и использует эти данные для перенаправления запроса серверу приложения. В xml файле указан IP-адрес/порт, который слушает сервер приложения. Веб-сервер, использует HTTP протокол для отправки полученного запроса на сервер приложения.

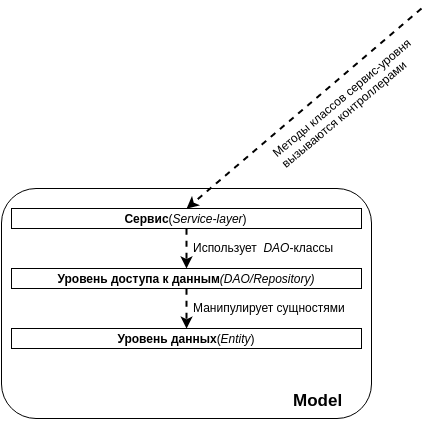
Внутри сервера приложения запрос передается определенному сервлету. Обработка запроса сервлетом производится в отдельном потоке. Сервлет представляет собой Java веб-компонент, управляемый контейнером, и способный предоставлять динамический контент. Аналогично и другим Java компонентам, сервлеты представляют собой платформонезависимые Java классы, компилируемые в Java байт код, который может быть динамически загружен и выполнен веб-сервером, поддерживающим Java технологии. Контейнером является расширение веб-сервера, которое предоставляет функциональность сервлетов. Сервлеты взаимодействуют с клиентами через запрос/ответ, реализуемые на уровне контейнера сервлетов.

**Обзор Spring MVC**

Архитектура веб-приложений строится с использованием паттерна MVC (Model-View-Controller):

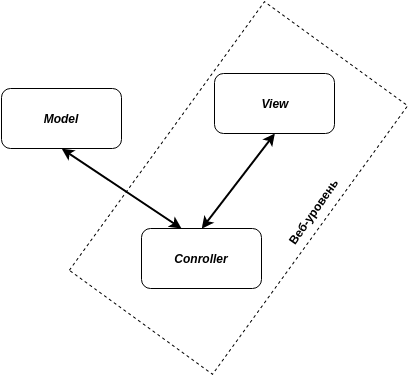


Элемент Model представляет собой данные (класс-сущности и уровень доступа к ним) и механизмы манипуляции этими данными (сервис-уровень):



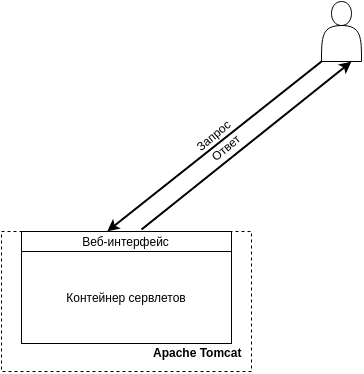
Ранее мы сами непосредственно в классе **Main** инициализировали контекст, получали бин класса сервис-уровня и вызывали его методы. В MVC-архитектуре контекст будет инициализироваться автоматически при разворачивании приложения на веб-сервере, а использовать методы классов сервис-уровня будет контроллер.

Рассмотрим организацию веб-уровня:



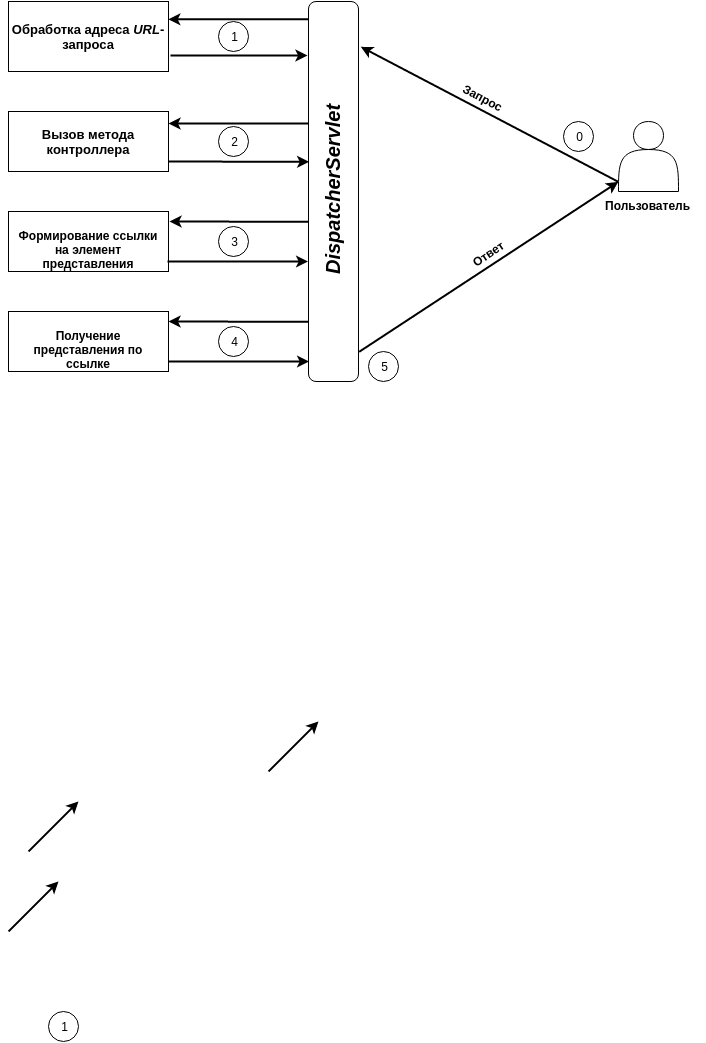
Конечная цель разработки веб-уровня — возможность взаимодействия пользователей с приложением непосредственно через свой браузер. Но кроме непосредственной реализации веб-уровня необходимо разработать соответствующую инфраструктуру, чтобы любой пользователь мог взаимодействовать с приложением через http/https-протокол.

Для сетевого взаимодействия с пользователем используются классы, расширяющие интерфейс **Servlet** и его дочерние классы. Такой класс способен взаимодействовать с пользователем по принципу «запрос — ответ». Но то, как обращаться с классами-сервлетами, «знает» только контейнер сервлетов. Еще он обеспечивает пользователям возможность делать запрос к сервлетам, выступая в роли веб-сервера. Один из самых популярных контейнеров сервлетов — **Apache Tomcat**.



**Обработка запросов в Spring MVC**

При использовании Spring MVC нет необходимости писать собственные сервлеты. Spring MVC предоставляет единственный «умный» сервлет **DispatcherServlet**, который поможет направить запрос пользователя соответствующему классу-контроллеру, созданному разработчиком. **DispatcherServlet** является входным контроллером. Процесс обработки пользовательского запроса в Spring MVC выглядит следующим образом:



Рассмотрим каждый из этапов:

* **Шаг 0.** Пользователь делает запрос, который содержит URL-адрес запроса и, возможно, какие-то данные.
* **Шаг 1.** Все запросы поступают на **DispatсherServlet**, который обязан перенаправить их конкретному контроллеру. Контроллеров может быть много, поэтому **DispatcherServlet** обращается к **HandlerMapping**, который на основании URL-строки запроса возвращает информацию о классе контроллера и его методе, который необходимо вызвать.
* **Шаг 2.** **DispatcherServlet** вызывает метод контроллера, передавая в него класс объекта **Model**. Метод, как правило, возвращает имя представления и может добавлять в объект класса **Model** данные, которые необходимо в дальнейшем передать пользователю.
* **Шаг 3.** На данном этапе у **DispatcherServlet** могут быть данные, которые являются результатом работы метода контроллера, и имя отображения. Дальнейшие действия — добавить эти данные в представление, но для начала нужно получить ссылку на представление. Для формирования ссылки **DispatcherServlet** обращается к **ViewResolver**, который на основании выбранного разработчиком правила формирует полную ссылку на представление (например, JSP или HTML) и возвращает ее **DispatcherServlet**.
* **Шаг 4. DispatcherServlet** получает конкретное представление по сформированной ранее ссылке (пути), добавляет в представление данные, отрисовывает его в HTML-страницу (но не обязательно).
* **Шаг 5. DispatcherServlet** возвращает ответ пользователю, и он отображается в браузере.

**Контроллеры**

Вызов объектов классов, представляющих собой бизнес-логику приложения, происходит в определенном контроллере. Контроллеры являются бинами Spring MVC.

Чтобы объявить контроллер, необходимо сделать следующее:

* добавить аннотацию **@Controller** уровня класса;
* к методу контроллера добавить аннотацию **@RequestMapping**, которая в качестве параметра принимает адрес (или часть адреса), содержащегося в пользовательском запросе, и метод запроса.

Рассмотрим пример объявления простого контроллера:

|  |
| --- |
| @Controller  @RequestMapping("/home")  public class HomeController {    @RequestMapping(value="/start", method=RequestMethod.GET)    public String hello(Model uiModel) {      return "home";    }  } |

Когда запустим приложение c подобным контроллером в контейнере сервлетов и перейдем по адресу <http://localhost:8080/app/home/start>, на экране браузера отобразится приветствие. В данном случае метод контроллера не добавляет никаких данных, а просто возвращает имя стандартного представления. Оно содержится в **/WEB-INF/templates/home.html**. Возврат имени производится методом **hello**, который помечен аннотацией **@RequestMapping** с указанием значений атрибутов **value** и **method**. Указание этих значений говорит о том, что данный метод вызывается только при GET-запросах.

URL-запрос для обращения к методу контроллера формируется следующим образом:

http://localhost:8080/app/home/start

[http://localhost]:[8080]/[app]/[home]/[start]

[хост]:[порт]/[название проекта]/[путь из @RequestMapping класса]/[ путь из @RequestMapping метода]

Рассмотрим второй вариант:

|  |
| --- |
| @Controller  @RequestMapping("/home")  public class HomeController {    @RequestMapping(value="/start", method = RequestMethod.GET)    public String hello(Model uiModel) {      return "home";    }      @RequestMapping(value="/start", method = RequestMethod.GET) // ошибка    public String hello2(Model uiModel) {      return "home";    }  } |

В данном случае получим ошибку, так как **DispatcherServlet** не сможет выяснить, какой метод вызвать при URL-запросе <http://localhost:8080/app/home/start>.

Все описанные выше методы просто возвращали представление. Теперь попытаемся передать данные на HTML-страницу. Сначала ее необходимо отредактировать следующим образом:

|  |
| --- |
| <!DOCTYPE html>  <html xmlns:th="http://www.thymeleaf.org">      <head>          <title>Home</title>      </head>      <body>          <h1 th:text="**'**Hello, **'** + ${name}" />      </body>  </html> |

В приведенном выше фрагменте страницы следует обратить внимание на запись "**'**Hello, **'** + ${name}". Страница будет ожидать передачи параметра с названием **name** из метода **hello**. Тогда метод контроллера будет выглядеть так:

|  |
| --- |
| @Controller  @RequestMapping("/home")  public class HomeController {    @RequestMapping(value = "/start", method = RequestMethod.GET)    public String hello(Model uiModel){      uiModel.addAttribute("name", "World");      return "home";    }  } |

В метод контроллера ссылку на объект класса **Model** передает **DispatcherServlet**. В методе контроллера происходит добавление в объект определенных данных с помощью метода **addAttribute(...)**, который принимает два параметра:

* **name** — имя объекта, которое будет использоваться для отображения данного объекта на HTML-странице с помощью EL;
* **object** — ссылка на объект.

Перейдя по соответствующему URL-адресу на страницу, увидим сообщение Hello World.

Строковые значения клиент может передавать прямо в строке URL-запроса с помощью аннотации **@PathVariable**:

|  |
| --- |
| @Controller  @RequestMapping("/home")  public class HomeController {    @RequestMapping(value = "/start/{name}", method = RequestMethod.GET)    public String hello(Model uiModel, @PathVariable(value="name") String name){      uiModel.addAttribute("name", name);      return "home";    }  } |

В таком случае все, что следует после **start/**, заносится в переменную **name**, которая передается в качестве параметра в метод контроллера:

http://localhost:8080/app/home/start/**bob**

Посмотрим, как контроллер взаимодействует с сервис-уровнем. Предположим, что необходимо, чтобы на главной странице сайта отображались имена всех авторов, которые есть в базе данных.

|  |
| --- |
| @Controller  @RequestMapping("/authors")  public class AuthorsController {    private AuthorsService authorsService;    @Autowired    public AuthorsService setAuthorsService (AuthorsService authorsService) {      this.authorsService = authorsService;    }      @RequestMapping(value = "/", method = RequestMethod.GET)    public String showAllAuthors(Model uiModel){      List<Author> authors = authorsService.getAll();      uiModel.addAttribute("authors", authors);      return "home";    }  } |

Страница **home.html** будет иметь следующий вид:

|  |
| --- |
| <!DOCTYPE html>  <html xmlns:th="http://www.thymeleaf.org">      <head>          <title>Home</title>      </head>      <body>          <h1>All authors</h1>          <p th:each="author : ${authors}" th:text="${author.firstName}" />      </body>  </html> |

В данном случае в браузере отобразятся имена всех авторов.

Передача параметров в пути запроса дает возможность производить поиск данных по определенному критерию. Теперь представим, что необходимо вывести информацию об авторе с запрашиваемым id.

Контроллер будет иметь следующий вид:

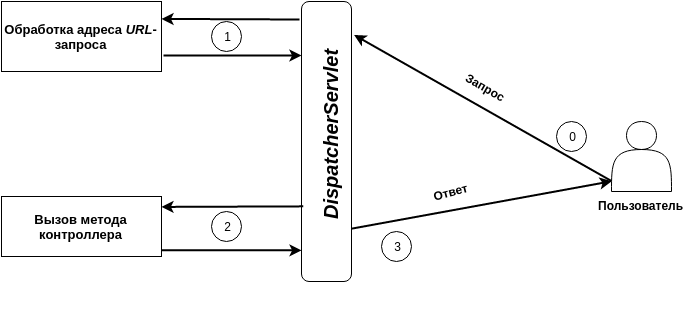
|  |
| --- |
| @Controller  @RequestMapping("/authors")  public class AuthorsController {    private AuthorsService authorsService;    @Autowired    public void setAuthorsService(AuthorsService authorsService) {      this.authorsService = authorsService;    }      @RequestMapping(value="/{id}", method = RequestMethod.GET)    public String home(Model uiModel, @PathVariable(value="id") Long id) {      Author author = authorsService.get(id);      uiModel.add("author", author);      return "home";    }  } |

Изменив имя переменной языка выражений в **home.html** и перейдя по адресу <http://localhost:8080/app/authors/1>, мы получим имя автора с id=1.

Кроме того, можно передавать возвращаемый результат напрямую в теле ответа:

|  |
| --- |
| @Controller  @RequestMapping("/authors")  public class AuthorsController {    private AuthorsService authorsService;    @Autowired    public void setAuthorsService(AuthorsService authorsService) {      this.authorsService = authorsService;    }      @RequestMapping(value="/{id}", method = RequestMethod.GET)    @ResponseBody    public String hello(@PathVariable(value="id") Long id){      Author author = authorsService.get(id);      return author.getFirstname();    }  } |

В данном случае последовательность запросов будет следующая:



Последовательность вызовов:

* **Шаг 0.** Пользователь делает запрос, который содержит URL-адрес запроса и, возможно, данные.
* **Шаг 1.** Все запросы поступают на **DispatсherServlet**, который обязан перенаправить их конкретному контроллеру. Их может быть много, поэтому **DispatcherServlet** обращается к другим классам. На основании URL-строки запроса они возвращают информацию о классе контроллера и его методе, который необходимо вызвать.
* **Шаг 2.** Происходит вызов метода соответствующего контроллера, который возвращает результат для его отображения клиенту.
* **Шаг 3.** Возвращенный в шаге 2 результат упаковывается в тело ответа и отправляется клиенту.

С помощью такого подхода можно возвращать данные в JSON-формате (рассмотрим далее).

**Работа с формами**

Рассмотрим передачу группы параметров, ассоциирующихся с сущностью в теле POST-запроса. Предположим, что необходимо дать клиенту возможность добавлять новых авторов. Тогда HTML будет выглядеть так:

|  |
| --- |
| <!DOCTYPE html>  <html xmlns:th="http://www.thymeleaf.org">     <head>        <title>Add new author</title>     </head>     <body>         <form th:action="@{/authors/form}" th:object="${author}" method="post">             First Name: <input type="text" th:field="\*{firstName}"/>             <br>             Last Name: <input type="text" th:field="\*{lastName}"/>             <br>             e-mail: <input type="text" th:field="\*{email}"/>             <br>             <button type="submit">Save</button>         </form>     </body>  </html> |

Атрибут **th:object** тега **form** указывает на имя объекта, который был добавлен через **uiModel.addAttribute** (**String name**, **Object object**), а атрибуты **th:field** тега **input** служат для получения доступа к полям объекта.

В общем случае для добавления сущности с помощью данной HTMLнеобходимо выполнить следующие шаги:

* **Шаг 1.** Создать пустой объект-сущность, добавить его в объект модели и вернуть имя представления, в котором содержится данная форма. Тогда **DispatcherServlet** добавит объект не по имени переменной **EL**, а по наименованию, указанному в атрибуте **th:object** тега **form**. Данный механизм предоставляет доступ к атрибутам объекта с возможностью их изменения.
* **Шаг 2.** Получить POST-запрос, в котором содержатся данные, ассоциированные с конкретными полями. **DispatcherServlet** внедрит эти значения в поля объекта.
* **Шаг 3.** В методе контроллера, принимающего POST-запрос и объект класса сущности, добавляемой в базу данных, вызвать метод сервиса, который сохраняет объект в БД.

Контроллер, обеспечивающий данный процесс:

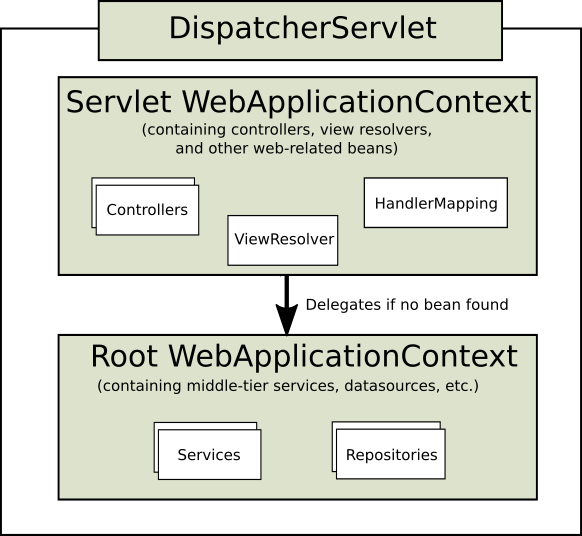
|  |
| --- |
| @Controller  @RequestMapping("/authors")  public class AuthorsController {    private AuthorsService authorsService;    @Autowired    public AuthorsService setAuthorsService(AuthorsService authorsService) {      this.authorsService = authorsService;    }      @RequestMapping(params = "form", method = RequestMethod.GET)    public String getForm(Model uiModel){      Author author = new Author();      uiModel.addAttribute("author", author);      return "home";    }    @RequestMapping(params = "form", method = RequestMethod.POST)    public String create(Author author) {      authorsService.save(author);      return "home";    }  } |

Метод **getForm(...)** делает все, что описано в шаге 1, а метод **create(...)** соответствует шагу 2 и 3. Метод **create(...)** вызывается после того, как клиент заполнил все поля формы и отправил ее POST-запроcом. В метод **create(...) DispatcherServlet** передает объект, полям которого присвоены значения, полученные из формы.

**Контекст Spring MVC**

В классе контроллеров все сервисы внедрялись с помощью аннотации **@Autowired**. Это значит, что у веб-уровня и остальной части приложения один общий контекст. Контроллеры — эти бины Spring, которые являются синглтонами и предназначены для взаимодествия с пользователем через **DispatcherServlet**.

В общем случае контекст в веб-приложениях выглядит следующим образом:



Компоненты Spring MVC, такие как **Controllers**, созданные разработчиком, а также **ViewResolver** и **HandlerMapping**, относятся к контексту сервлета. Остальные компоненты, созданные ранее, — к корневому контексту приложения. Тем не менее контекст сервлета и главный контекст приложения объединяются.

**Очень коротко об обработке запросов**

При первом прочтении этот пункт можно пропустить. А вот после решения некоторого количества практических задач, очень полезно вернуться и пройтись по материалу. Частично материал пересекается, с описанным выше, но тут он сконцентрирован, чтобы не было необходимости прыгать по отдельным главам.

Давайте в одном пункте разберем все что касается формирования запросов и их перехвата с помощью контроллеров. Для дальнейшего разбора важно зафиксировать **context-path** нашего web-приложения, пусть это будет [**http://localhost:8189/application**](http://localhost:8189/application). То есть все запросы будем рассматривать относительно этого пути

Если вы вводите некий url в адресной строке браузера, то вы посылаете **GET-запрос** на сервер. Например, <http://localhost:8189/application/index>. Часть [http://localhost:8189/application](http://localhost:8189/application/index) означает что запрос полетит в наше приложение, но внутри приложения мы должны обработать часть /index. Как это сделать:

|  |
| --- |
| @RequestMapping(value = **"/index"**, method = RequestMethod.***GET***)  **public** String showIndexPage() {  **return "index\_page"**;  } |

**@RequestMapping** означает что если на адрес [[http://localhost:8189/application]**/index**](http://localhost:8189/application%5d/index) прилетит GET запрос, то его будет обрабатывать именно этот метод. В квадратных скобках будем дублировать context-path чтобы был виден полный путь. Если туда прилетит запрос POST, PUT, DELETE, то метод его не перехватит. В проекте НЕ МОЖЕТ быть нескольких методов, обрабатывающих один и тот же запрос. При старте, Spring производит маппинг url’ов к тем методам, которые их обрабатывают, при возникновении ситуации, когда два разных метода собираются обрабатывать один и тот же URL+Http.Method, Spring выдаст ошибку.

|  |
| --- |
| @RequestMapping(value = **"/index"**, method = RequestMethod.***GET***)  **public** String showIndexPage() {  **return "index\_page"**;  }  @RequestMapping(value = **"/index"**, method = RequestMethod.***GET***)  **public** String anotherIndexPage() {  **return "index\_page"**;  } |

Если ситуация с двумя обработчиками GET /index недопустима, то совершенно спокойно может существовать следующая ситуация:

|  |
| --- |
| @RequestMapping(value = **"/index"**, method = RequestMethod.***GET***)  **public** String showGetIndexPage() {  **return "index\_page"**;  }  @RequestMapping(value = **"/index"**, method = RequestMethod.***POST***)  **public** String doSomethingWithPostIndex() {  **return "redirect:/index"**;  } |

В данном случае никакой неоднозначности нет. Один метод перехватит GET-запрос, второй - POST-запрос. Если в аннотации @RequestMapping HttpMethod не указан, то будет перехватываться как POST, так и GET запрос, пример:

|  |
| --- |
| @RequestMapping(value = **"/index"**)  **public** String showIndexPage() {  **return "index\_page"**;  } |

Более короткая версия аннотации @RequestMapping(value = **"/index"**, method = RequestMethod.***GET***) может записана как @GetMapping(**"/index"**), **абсолютно никакой разницы** в логике работы между этими формами нет. Так же как и нет разницы в @RequestMapping(value = **"/index"**, method = RequestMethod.***POST***)и @PostMapping(**"/index"**).

Теперь когда мы посмотрели как метод понимает какой запрос он должен обработать, давайте посмотрим что мы можем из запроса вытащить. Вот несколько типичных GET-запросов:

|  |
| --- |
| [[http://localhost:8189/application]**/items/info?id=5&title=Milk&price=80**](http://localhost:8189/application/items/info?id=5&title=Milk&price=80)  [[http://localhost:8189/application]**/users/profile?id=1**](http://localhost:8189/application/users/profile?id=1)  [[http://localhost:8189/application]**/video?id=13432&category=edu**](http://localhost:8189/application/users/profile?id=1) |

В приведенных выше url’ах содержаться дополнительные параметры, которые идут после символа ? и разделены &. При обработке таких запросов, мы можем вытащить параметры и использовать их для обработки запроса.

|  |
| --- |
| @GetMapping(value = **"/items/info"**) **public** String getItemInfo(@RequestParam(name = **"id"**) Long id, @RequestParam(name = **"title"**) String title, @RequestParam(name = **"price"**) **int** price) {    *// ... тут какая-то логика ...* } |

@RequestParam означает что в запросе у нас есть параметры, значения которых мы хотим получить. Из url эти значения приходят в виде строк, но Spring можем их автоматически преобразовывать в нужный нам формат. Примере выше id был завернут в Long, а price в Integer. Если имя параметра в запросе и имя атрибута метода совпадают, то можно не указывать (name = …).

|  |
| --- |
| @GetMapping(value = **"/items/info"**) **public** String getItemInfo(@RequestParam Long id, @RequestParam String title, @RequestParam **Integer** price) {    *// ... тут какая-то логика ...* } |

Spring с помощью аннотации @RequestParam не делает ничего кроме вытаскивания необходимого параметра из запроса и преобразования в нужный тип. Если вы указали что ждете в запросе набор параметров: id, title и price, а запрос приходит без этих параметров, то получите исключение.

|  |
| --- |
| GET [http://localhost:8189/application]**/items/info?id=1&title=box**  @GetMapping(value = **"/items/info"**) **public** String getItemInfo(@RequestParam Long id, @RequestParam String title, @RequestParam **Integer** price) {    *// ... тут какая-то логика ...* } |

Например вот в таком запросе не хватает параметра price. Если вы считаете что параметры не обязательно будут присутствовать в запросе, то такие параметры можно пометить на необязательные с помощью @RequestParam(**required = false**).

|  |
| --- |
| GET [http://localhost:8189/application]**/items/info?id=1&title=box**  @GetMapping(value = **"/items/info"**) **public** String getItemInfo(@RequestParam(required = false) Long id, @RequestParam(required = false) String title, @RequestParam(required = false) **Integer** price) {    *// ... тут какая-то логика ...* } |

Теперь если в запросе не хватает части параметров, то вместо них вы просто получите null.

Если же работаете с POST запросами, то вы также можете вытаскивать RequestParam’ы, другое дело что их не будет видно в адресной строке, поскольку они зашиваются в тело POST запроса. Пример:

|  |
| --- |
| POST [http://localhost:8189/application]**/items/info**  @PostMapping(value = **"/items/info"**) **public** String doItemsPost(@RequestParam(required = false) Long id, @RequestParam(required = false) String title, @RequestParam(required = false) **Integer** price) {    *// ... тут какая-то логика ...* } |

Помимо получения параметров, данные можно вытаскивать прямо из url. Допустим у нас есть запрос вида: [[http://localhost:8189/application]/user/**12**/course/**10**](http://localhost:8189/application%5d/user/12/course/10), и на местах чисел 12 и 10 могут попадаться различные значения, в таком случае мы можем получать эти значения с помощью @PathVariable.

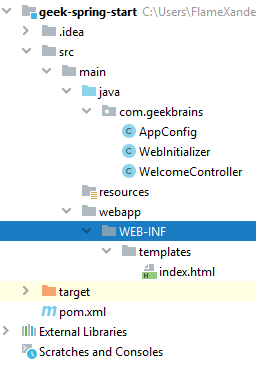
|  |
| --- |
| @GetMapping(value = **"/user/{user\_id}/course/{course\_id}"**) **public** String pathDemo(@PathVariable(name = **"user\_id"**) Long userId, @PathVariable(name = **"course\_id"**) Long courseId) {    *// ... тут какая-то логика ...* } |

То есть с помощью {имя\_переменной} мы указываем части запроса, которые хотим вытащить и записать в атрибуты метода.

**Практика**

**Создание проекта**

Необходимо создать проект с помощью **Maven**: **«mvn archetype:generate -DgroupId=com.geekbrains -DartifactId=geek-spring-start -DarchetypeArtifactId=maven-archetype-webapp D-archetypeVersion=1.4 -DinteractiveMode=false»**. После генерации проекта можно удалить стандартные **web.xml**, поскольку вся настройка будет организована через **JavaConfig**. И в папку main добавить папку java, нажать на папке java правой кнопкой и выполнить Mark Directory As -> Source Root. Итоговая структура базового проекта примет такой вид:



**Настройка проекта**

Для разработки веб-приложений с использованием Spring MVC файл **pom.xml** необходимо скорректировать:

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0"           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  http://maven.apache.org/maven-v4\_0\_0.xsd">    <modelVersion>4.0.0</modelVersion>    <groupId>com.geekbrains</groupId>    <artifactId>geek-spring-start</artifactId>    <packaging>war</packaging>    <version>1.0-SNAPSHOT</version>    <name>GeekSpringStart Maven Webapp</name>    <url>http://maven.apache.org</url>    <properties>      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>      <maven.compiler.source>1.8</maven.compiler.source>      <maven.compiler.target>1.8</maven.compiler.target>      <spring.version>5.1.0.RELEASE</spring.version>    </properties>    <dependencies>      <dependency>        <groupId>org.springframework</groupId>        <artifactId>spring-webmvc</artifactId>        <version>${spring.version}</version>      </dependency>      <dependency>        <groupId>org.springframework</groupId>        <artifactId>spring-test</artifactId>        <version>${spring.version}</version>      </dependency>      <dependency>        <groupId>ch.qos.logback</groupId>        <artifactId>logback-classic</artifactId>        <version>1.2.3</version>      </dependency>      <dependency>        <groupId>org.junit.jupiter</groupId>        <artifactId>junit-jupiter-engine</artifactId>        <version>5.3.1</version>        <scope>test</scope>      </dependency>      <dependency>        <groupId>org.hamcrest</groupId>        <artifactId>hamcrest-library</artifactId>        <version>1.3</version>        <scope>test</scope>      </dependency>      <dependency>        <groupId>javax.servlet</groupId>        <artifactId>javax.servlet-api</artifactId>        <version>3.1.0</version>        <scope>provided</scope>      </dependency>      <dependency>        <groupId>org.thymeleaf</groupId>        <artifactId>thymeleaf-spring5</artifactId>        <version>3.0.10.RELEASE</version>      </dependency>    </dependencies>    <build>      <finalName>java-web-project</finalName>      <plugins>        <plugin>          <groupId>org.apache.tomcat.maven</groupId>          <artifactId>tomcat7-maven-plugin</artifactId>          <version>2.2</version>          <configuration>            <url>http://localhost:8080/manager/text</url>            <username>geek</username>            <password>geek</password>            <path>/app</path>          </configuration>        </plugin>        <plugin>          <groupId>org.apache.maven.plugins</groupId>          <artifactId>maven-surefire-plugin</artifactId>          <version>2.22.0</version>        </plugin>        <plugin>          <groupId>org.apache.maven.plugins</groupId>          <artifactId>maven-war-plugin</artifactId>          <version>3.2.2</version>        </plugin>      </plugins>    </build>  </project> |

* **spring-web** — обеспечивает взаимодействие с HTTP, включает в себя некоторые фильтры и другие веб-компоненты.
* **spring-webmvc** — является реализацией Spring MVC и зависит от **spring-web**.
* **thymeleaf** — шаблонизатор для обработки html файлов.

**Конфигурирование**

Создадим класс конфигурации приложения.

|  |
| --- |
| @EnableWebMvc  @Configuration  @ComponentScan(**"com.geekbrains"**)  **public class** AppConfig **implements** WebMvcConfigurer {     @Override  **public void** addResourceHandlers(ResourceHandlerRegistry registry) {       registry.addResourceHandler(**"/resources/\*\*"**).addResourceLocations(**"/resources/"**);     }     @Bean  **public** SpringResourceTemplateResolver templateResolver() {         SpringResourceTemplateResolver templateResolver = **new** SpringResourceTemplateResolver();         templateResolver.setPrefix(**"/WEB-INF/templates/"**);         templateResolver.setSuffix(**".html"**);  **return** templateResolver;     }     @Bean  **public** SpringTemplateEngine templateEngine() {         SpringTemplateEngine templateEngine = **new** SpringTemplateEngine();         templateEngine.setTemplateResolver(templateResolver());  **return** templateEngine;     }     @Bean  **public** ThymeleafViewResolver thymeleafViewResolver() {         ThymeleafViewResolver thymeleafViewResolver = **new** ThymeleafViewResolver();         thymeleafViewResolver.setTemplateEngine(templateEngine());         thymeleafViewResolver.setCharacterEncoding(**"UTF-8"**);  **return** thymeleafViewResolver;     }  } |

Этот класс — обычный **JavaConfig**, в котором объявляются бины Spring. Разница лишь в том, что они относятся к веб-уровню приложения.

В классе объявлены следующие аннотации:

* **@Configuration** — указывает на то, что данный класс является конфигурацией;
* **@EnableWebMvc** — включает поддержку аннотации MVC-компонентов (например, @Controller);
* **@ComponentScan** — в данном случае указывает на пакет, в котором хранятся класс-контроллеры;
* **@Import** — указывается класс корневой конфигурации приложения. Благодаря данной аннотации происходит объединение контекстов.

В классе есть два метода:

1. **addResourceHandlers(...)** — добавляет обработчик ресурсов. Метод принимает объект класса **ResourceHandlerRegistry** и добавляет шаблон пути и локацию. Другими словами, на все запросы **/resources/\*\*** будет вызываться не контроллер, созданный разработчиком, а возвращаться указанный в запросе файл (например, **.css** или **.js**).
2. **thymeleafViewResolver()** — создает и настраивает бин, который является тем самым **ViewResolver**. Вспомним: контроллер возвращает только строку имени html-страницы, а **DispatcherServlet** обращается к данному бину, который формирует полный путь к представлению, прибавляя к его имени параметры, указанные в методах **setPrefix** и **setSuffix**.

Посмотрим на класс для настройки **DispatcherServlet**:

|  |
| --- |
| public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {   @Override   protected Class<?>[] getRootConfigClasses() {     return null;   }    @Override   protected Class<?>[] getServletConfigClasses() {     return new Class[]{AppConfig.class};   }    @Override   protected String[] getServletMappings() {     return new String[]{"/"};   }    @Override   protected Filter[] getServletFilters() {     *// Создание фильтра кодировки, который позволит работать с русскими*     *// символами*     CharacterEncodingFilter characterEncodingFilter = new CharacterEncodingFilter();     characterEncodingFilter.setEncoding("UTF-8");     characterEncodingFilter.setForceEncoding(true);     *// Создание фильтра, который добавляет поддержку HTTP-методов (например*     *// таких, как PUT), необходимых для REST API*     HiddenHttpMethodFilter httpMethodFilter = new HiddenHttpMethodFilter();     return new Filter[]{characterEncodingFilter, httpMethodFilter};   } } |

Данный класс расширяет абстрактный класс **AbstractAnnotationConfigDispatcherServletInitializer**, в котором автоматически вызываются методы **createRootApplicationContext()** и **createServletApplicationContext()**. Они используют переопределенные нами методы для объявления сервлета и формирования общего контекста.

**Создание представления**

Создадим представление, с помощью которого протестируем работоспособность веб-уровня. Реализуем это представление в виде простого **HTML**.

Представление создадим в **src/main/webapp/WEB-INF/templates** и назовем **index.html**:

|  |
| --- |
| <html>   <body>     <h1 th:text="${message}" />     <h2>Spring</h2>   </body> </html> |

**Создание контроллера**

Задача контроллера будет заключаться в передаче сообщения и возврате имени представления **index.html**. Класс контроллера будет иметь следующий вид:

|  |
| --- |
| @Controller public class WelcomeController {   @GetMapping("/")   public String index(Model model) {     model.addAttribute("message", "Hello World");     return "index";   } } |

**Установка Apache Tomcat 9**

Для запуска проекта необходимо установить и запустить Apache Tomcat 7, 8 или 9-й версии. После того, как **Apache Tomcat** будет установлен на компьютере, надо найти файл **...\Apache Software Foundation\Tomcat 9.0\conf\users.xml** и добавить туда пользователя:

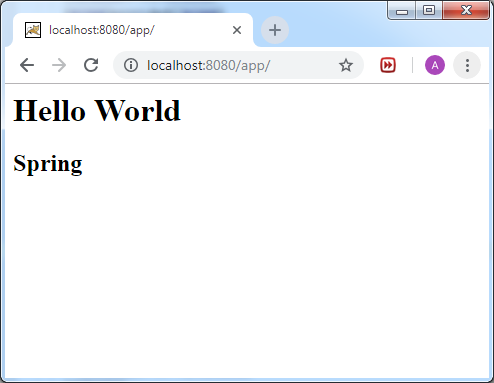
|  |
| --- |
| <tomcat-users xmlns="http://tomcat.apache.org/xml"               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"               xsi:schemaLocation="http://tomcat.apache.org/xml tomcat-users.xsd"               version="1.0"> <role rolename="manager-gui"/> <role rolename="manager-script"/> <user username="geek" password="geek" roles="admin-gui,manager-gui,tomcat,manager-jmx,manager-script" /> </tomcat-users> |

Это необходимо, чтобы можно было деплоить проект из Maven. Логин/пароль и путь к приложению уже прописать в **pom.xml**, как было показано выше.

**Запуск приложения из IntelliJ IDEA с помощью Maven**

Для запуска необходимо в терминале выполнить команду: **mvn tomcat7:deploy**.

Для отображения результата переходим по адресу <http://localhost:8080/app/>. Если все сделано правильно, в браузере отобразится следующая страница:



**Практическое задание**

1. Разобраться с примером проекта на Spring MVC.
2. Создать класс Товар (Product), с полями **id**, **title**, **cost**.
3. Товары необходимо хранить в репозитории (класс, в котором в виде **List<Product>** хранятся товары). Репозиторий должен уметь выдавать список всех товаров и товар по id.
4. Сделать форму для добавления товара в репозиторий и логику работы этой формы.
5. Сделать страницу, на которой отображаются все товары из репозитория.

**Дополнительные материалы**

1. [Официальная документация по Spring Web MVC](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html)

**Используемая литература**

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://developer.mozilla.org/ru/docs/Learn/%D0%A7%D1%82%D0%BE_%D1%82%D0%B0%D0%BA%D0%BE%D0%B5_%D0%B2%D0%B5%D0%B1_%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80>
2. Крейг Уоллс. Spring в действии

**Доступ к данным в Spring. Часть 1**

Hibernate. Понятие сущности. Объектно-реляционное отображение. Отображение связей «один ко многим», «один к одному», «многие ко многим». Контекст постоянства. Менеджер сущностей. JPQL. Доступ к атрибутам. Каскадные операции. Управление транзакциями.

**Оглавление**

[Введение](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.2et92p0)

[Hibernate в Spring-приложении](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.tyjcwt)

[Hibernate и JPA](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.3dy6vkm)

[Понятие сущности и объектно-реляционное отображение](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.1t3h5sf)

[Отображение связей](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.4d34og8)

[Один ко многим](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.2s8eyo1)

[Один к одному](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.17dp8vu)

[Многие ко многим](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.26in1rg)

[Доступ к атрибутам](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.34bbaodz4vuu)

[Каскадные операции](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.t3s94u4fofni)

[Контекст постоянства и EntityManager](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.74ua8j3aoyh2)

[JPQL](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.dw9pst7acio4)

[Обзор синтаксиса](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.vonyuhb7dwsj)

[Запросы](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.5u8ferg6iqq5)

[Динамические запросы](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.gpndtns1dkw5)

[Именованные запросы](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.njktc65d0gx)

[Аннотации](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.5kpen3y86lpv)

[Оптимистическое управление параллельным доступом](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.dtet2vlq0twh)

[Пессимистические блокировки](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.d02739p06ivx)

[Практическое задание](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.lnxbz9)

[Дополнительные материалы](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.35nkun2)

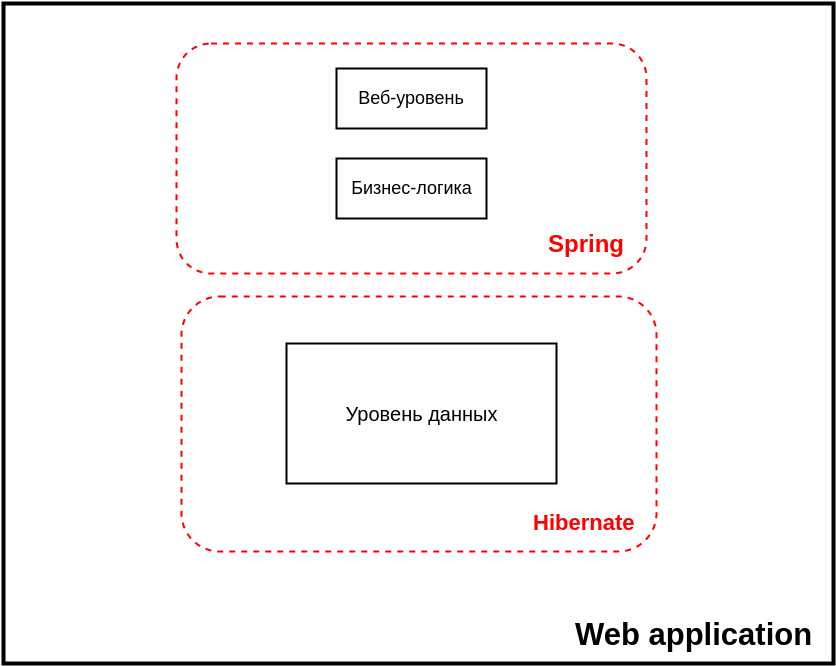
[Используемая литература](https://docs.google.com/document/d/1Cv8ILm2-2fXT-5QwVf-N3HKwqAi2gNI5VCGoiwuOWGs/edit#heading=h.1ksv4uv)

**Введение**

Spring предоставляет собственные средства для взаимодействия с СУБД. Ядром этого процесса является класс **JdbcTemplate**, но он довольно низкоуровневый и избавляет разработчика только от написания кода обработки ошибок и закрытия соединения. При создании серьезных приложений этого недостаточно, так как разработчику придется самостоятельно писать код для объектно-реляционного отображения. Для решения данной проблемы существуют ORM-библиотеки, самая популярная из которых — Hibernate.

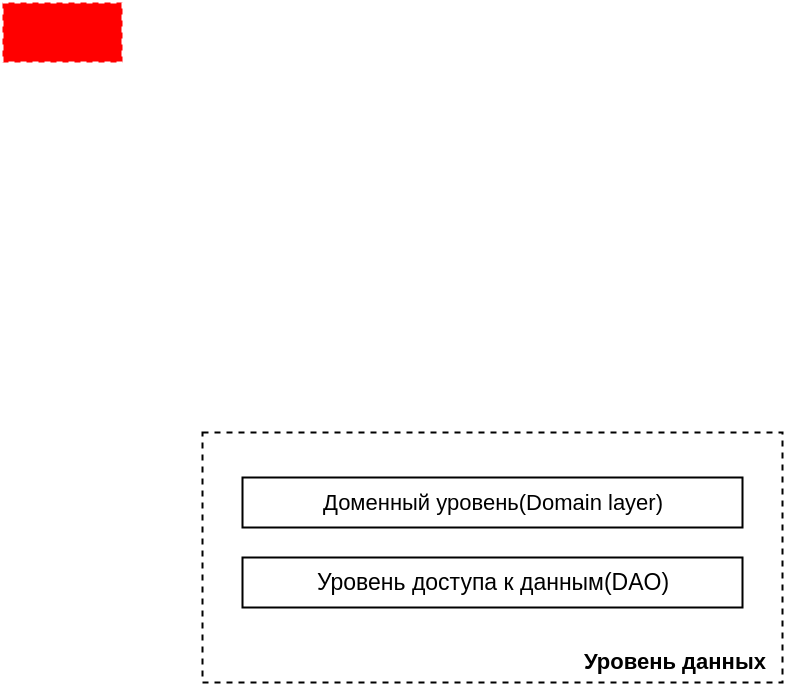
**Hibernate в Spring-приложении**

В предыдущих уроках мы часто упоминали компонент (бин) Spring. Теперь отойдем от темы Spring и изучим Hibernate. Это связано с тем, что практически во всех веб-приложениях, написанных с использованием Spring, для обеспечения взаимодействия с СУБД используется Hibernate, а средства фреймворка применяются для написания бизнес-логики и веб-уровня.



Уровень данных можно разделить на следующие ступени:

* **DAO** — **Data Access Object** — абстрактный объект, предоставляющий доступ к данным, хранящимся в БД. На самом деле, классы DAO-уровня — объекты Spring, но эти они используют средства Hibernate;
* **Domain** — на данном уровне хранятся модели, то есть основные объекты мэппинга. Это то, во что будут преобразовываться данные из БД (сущности).



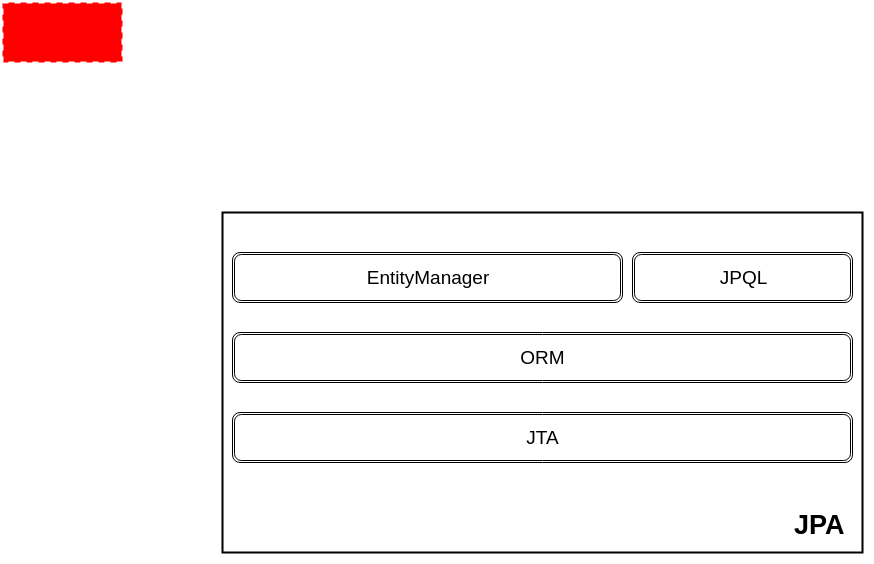
В данном уроке рассмотрим доменный уровень — способы его построения с использованием Hibernate. Работа же с базами данных с помощью Hibernate в Spring приложении будет рассмотрена на следующем занятии.

**Hibernate и JPA**

Изначально Hibernate развивалась как самостоятельная библиотека, не связанная со спецификацией Java EE. С третьей версии она приобрела поддержку JPA (Java Persistence API), фактически став ее реализацией: JPA определяет интерфейсы объектов для доступа к данным, а Hibernate их реализует. Поэтому в данном уроке, когда дело не касается конкретных реализаций, Hibernate и JPA будут взаимозаменяемыми понятиями.

Рассмотрим основные составляющие JPA:

* API для операций с данными, хранящимися в БД (вставки, удаления, изменения и других). Данный API описывается интерфейсом **EntityManager**;
* Объектно-реляционное отображение, которое описывает способы отображения объектов в данные, хранящиеся в БД;
* JPQL — Java Persistence Query Language — язык, позволяющий делать запросы к базе данных непосредственно из кода;
* JTA — Java Transaction API — механизмы работы с транзакциями.



При изучении доменного уровня нас интересуют способы отображения объектов в данные, хранящиеся в БД.

Для работы с PostgreSQL и подключения Hibernate к Maven проекту, необходимо добавить зависимости:

|  |
| --- |
| <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-core</artifactId>      <version>5.4.3.Final</version>  </dependency>  <dependency>      <groupId>org.postgresql</groupId>      <artifactId>postgresql</artifactId>      <version>42.2.5</version>  </dependency> |

Первая зависимость - ядро Hibernate, вторая - JDBC драйвер для работы с PostgreSQL. Далее, в ресурсы проекта добавляем файл настроек hibernate.cfg.xml.

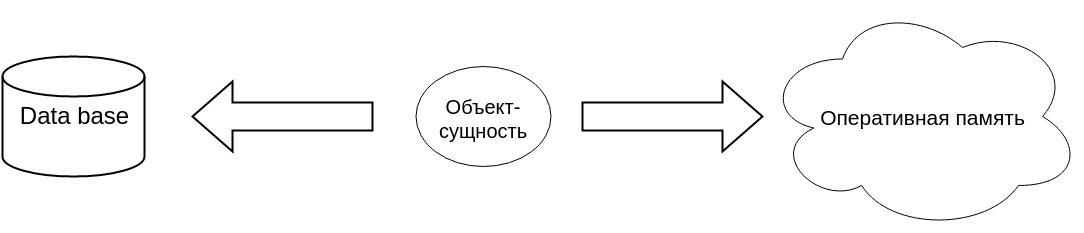
|  |
| --- |
| <!DOCTYPE hibernate-configuration PUBLIC          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"          "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  <hibernate-configuration>      <session-factory>          <property name="connection.driver\_class">org.postgresql.Driver</property>          <property name="connection.url">jdbc:postgresql://localhost:5432/postgres</property>          <property name="connection.username">postgres</property>          <property name="connection.password">admin</property>          <property name="connection.pool\_size">1</property>          <property name="dialect">org.hibernate.dialect.PostgreSQL94Dialect</property>          <property name="show\_sql">true</property>          <property name="current\_session\_context\_class">thread</property>          <mapping class="com.geekbrains.hibernate.Person"/>      </session-factory>  </hibernate-configuration> |

Давайте посмотрим что означают все эти свойства в конфигурационном файле.

* **connection.driver\_class** - имя класса JDBC драйвера;
* **connection\_url** - url для подключения к базе данных. В данном случае сервер базы данных развернут на локальной машине, на стандартном порту 5432, и имя базы данных - postgres;
* **connection.username**, **connection.password** - логин/пароль пользователя;
* **connection.pool\_size** - размер пула соединений;
* **dialect** - диалект SQL;
* **show\_sql** - включение/выключение логирования выполняемых SQL запросов;
* **current\_session\_context\_class** - указание области видимости сессии, в данном случае для каждого потока будет своя сессия;
* **mapping class** - перечисление хранимых классов.

**Понятие сущности и объектно-реляционное отображение**

Обычные объекты классов Java сохраняют свое состояние в оперативной памяти компьютера, но после завершения программы вся информация о них теряется. Объекты-сущности — это объекты Java, которые сохраняют свое состояние в базе данных, что обеспечивает их долгосрочное хранение. Чтобы сохранить состояние объекта, в БД должна располагаться таблица, соответствующая классу этого объекта. По аналогии со Spring, сущность — это объект, которым управляет класс **EntityManager**, относящийся к Hibernate.



Чтобы сущность могла сохранять свое состояние в базе данных, необходима возможность сохранения ее компонентов в БД. Объектами мэппинга могут быть следующие элементы:

* поля класса;
* класс;
* связи (отношения) между классами;
* и другие.

Чтобы объекты класса являлись сущностями, должны выполняться следующие условия:

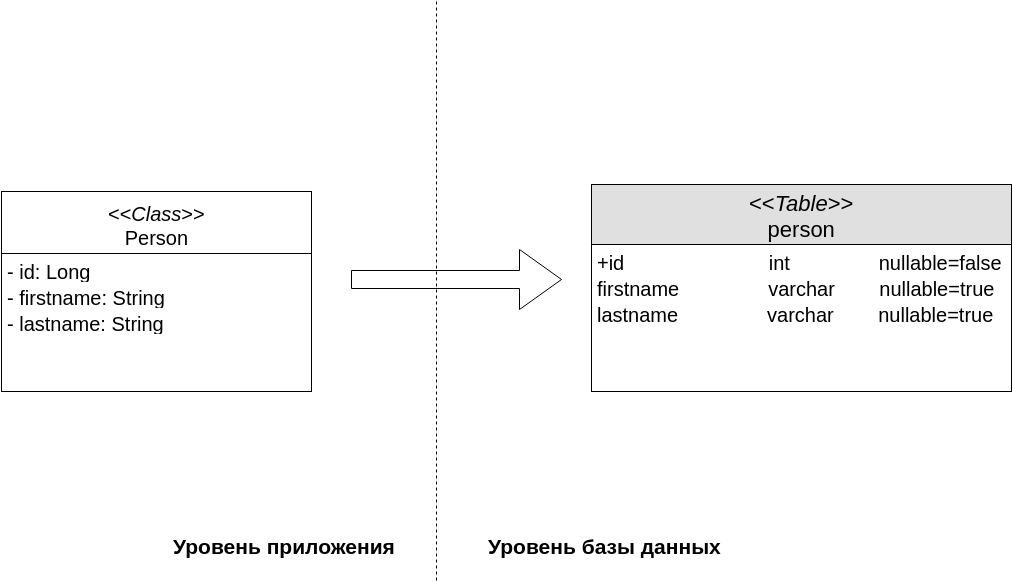
* наличие аннотации **@Entity**;
* наличие поля, в котором будет храниться уникальный идентификатор сущности. Оно должно быть снабжено аннотацией **@Id**;
* наличие конструктора без аргументов (конструктора по умолчанию);
* отсутствие модификатора **final**.

Порядок объявления сущности:

|  |
| --- |
| @Entity **public** **class** Person {    @Id   @GeneratedValue   **private** Long id;    **private** String firstname;    **private** String lastname;    *// Геттеры и сеттеры*   *// ...* } |

Прежде чем вдаваться в детали конфигурирования, отметим, что **Hibernate** (как и Spring) использует подход «конфигурация в порядке исключения». Фактически это означает, что если не задано иное, то будет использоваться поведение по умолчанию.

Для объектов вышеуказанного класса порядок отображения будет таким:



Данный процесс следует правилам:

1. Имя класса отображается в имя таблицы (**Person −> person**). Если таблица, в которую необходимо отобразить сущность, имеет другое имя, то следует использовать аннотацию **@Table** с указанием имени таблицы, в которую следует отобразить класс.
2. Имена атрибутов отображаются в имена столбцов (**firstname −> firstname**). Если имя столбца отличается от имени атрибута, необходимо использовать аннотацию **@Column** с указанием имени столбца.
3. Типы атрибутов класса отображаются в типы используемой СУБД. Этот процесс интуитивно понятен (например, **Long −> integer**), но отличается в различных СУБД.

С учетом вышесказанного предыдущее объявление сущности эквивалентно следующему коду:

|  |
| --- |
| @Entity @Table(name = **"person"**) **public** **class** Person {   @Id   @тип выб(strategy = GenerationType.IDENTITY)   @Column(name = **"id"**)   **private** Long id;    @Column(name = **"firstname"**)   **private** String firstname;    @Column(name = **"lastname"**)   **private** String lastname;    *// Геттеры и сеттеры*   *// ...* } |

Когда класс снабжен аннотацией **@Entity**, все его атрибуты по умолчанию будут отображаться в столбцы ассоциируемой таблицы. Но если нет необходимости в отображении какого-либо атрибута, следует применить к нему аннотацию **@Transient**.

Большинство аннотаций из данного листинга интуитивно понятны, но стоит подробнее рассмотреть **@GeneratedValue**.

**Важно отметить, что объект становится объектом-сущностью только после сохранения в базе данных.** Кроме того, любой объект-сущность должен иметь свой уникальный идентификатор. Но разработчику совсем не обязательно заботиться об этом. Как правило, это значение можно (и желательно) получить из самой базы данных. Большинство БД предоставляют собственные механизмы генерации значений id. Значение может инжектироваться в поле id объекта-сущности после его сохранения. Для этого необходимо использовать аннотацию **@GeneratedValue**. В зависимости от механизма генерации значений id в базе данных, атрибуту **strategy** аннотации **@GeneratedValue** присваиваются различные значения из перечисления **GenerationType**.

Атрибут **strategy** может иметь следующие значения:

* **GenerationType.SEQUENCE** — говорит о том, что значение id будет генерироваться с помощью sequence-генератора, созданного разработчиком в базе данных. При использовании данной стратегии необходимо дополнительно указывать имя генератора в атрибуте **name** аннотации **@GeneratedValue**;
* **GenerationType.IDENTITY** — указывает поставщику постоянства, что значение id необходимо получать непосредственно из столбца «id» таблицы, в которую мэппится данный объект-сущность;
* **GenerationType.AUTO** — предоставлялет Hibernate возможность самостоятельно выбрать стратегию для получения id, исходя из используемой СУБД;
* **GenerationType.TABLE** — говорит о том, что для получения значения id необходимо использовать определенную таблицу в БД, содержащую набор чисел.

Оптимальный подход — использование **GenerationType.IDENTITY**. Аннотация говорит Hibernate о том, что после сохранения объекта в базе данных необходимо получить значение из столбца, на который отображается атрибут id, и присвоить его объекту-сущности. А каким образом в этом столбце появится значение после вставки строки с информацией об объекте, остается на совести разработчика. Данный подход удобен при использовании СУБД PostgreSQL, в которой такому столбцу можно задать тип **serial** — и СУБД будет автоматически генерировать значения для данного столбца после вставки строки. В следующем уроке при рассмотрении контекста постоянства мы вернемся к данной теме.

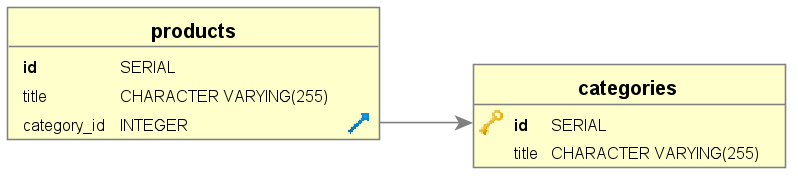
**Отображение связей**

На практике кроме простых атрибутов в сущностях присутствуют и связи между классами. Существует три вида связей: один к одному, один ко многим, многие ко многим. В базе данных организуются аналогичные связи между таблицами — за счет использования механизма внешних ключей. Давайте рассмотрим способы отображения связей.

Один ко многим

Связь «один ко многим» отображается с помощью аннотации **@OneToMany**, **@ManyToOne** и **@JoinColumn**. Представим, что у нас есть класс товара и категории (товаров). При этом каждый товар относится только к одной категории (*это условность в данном случае*), но одна категория может включать большое количество товаров. Данная ситуация относится к виду связи «один ко многим» (или «многие к одному»). В таблице данная связь отображается следующим образом:

|  |
| --- |
| CREATE TABLE categories (id serial, title varchar(255), PRIMARY KEY (id));  CREATE TABLE products (id serial, title varchar(255), category\_id integer REFERENCES categories (id)); |



Эта связь достигается использованием внешнего ключа **category\_id**, владельцем связи является таблица **products**. А вот как это будет организовано в коде:

|  |
| --- |
| @Entity  @Table(name = **"products"**)  **public class** Product {     @Id     @GeneratedValue     @Column(name = **"id"**)     private Long **id**;     @Column(name = **"title"**)     private String **title**;     @ManyToOne     @JoinColumn(name = **"category\_id"**)     private Category **category**;  *// ...*  }  @Entity  @Table(name = **"categories"**)  **public class** Category {     @Id     @Column(name = **"id"**)     @GeneratedValue     Long **id**;     @Column(name = **"title"**)     String **title**;     @OneToMany(mappedBy = **"category"**)     List<Product> **products**;  *// ...*  } |

Данное объявление подчиняется следующим правилам:

* для атрибутов обоих классов указывается аннотация **@ManyToOne** или **@OneToMany** в зависимости от стороны связи;
* для класса **Product**, который является владельцем связи, используется атрибут **name** аннотации **@JoinColumn**, которая указывает на столбец с внешним ключом в таблице (в данном случае @JoinColumn(name = **"category\_id"**));
* для класса **Category** в параметре **mappedBy** указывается название ассоциируемого с ним поля в классе-владельце **Product** (@OneToMany(mappedBy = **"category"**)).

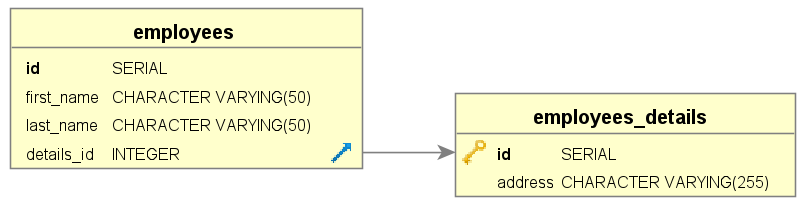
В данном случае связь между классами является двунаправленной, в отличие от таблиц, между которыми связь однонаправленная. Разработчик сам определяет, какой из вариантов использовать, но на практике оптимальна двунаправленная связь между классами. Использование двунаправленной связи избавляет нас от явного вызова кода запроса к базе данных.

Если в классе Category убрать @OneToMany(mappedBy = **"category"**), то у продукта мы сможем спросить его категорию, а вот у категории запросить список товаров уже нет (получается однонаправленная связь).

Один к одному

Представим что мы хотим хранить информацию о сотрудниках в таблице employees, но при этом для каждого сотрудника может быть заведена дополнительная “карточка” с информацией, которая будет мешать в основной таблице. Для этих дополнительных данных заведем таблицу employee\_details. Запросы на создание таблиц будет выглядеть следующим образом:

|  |
| --- |
| CREATE TABLE employees\_details (id serial, address varchar(255), PRIMARY KEY (id));  CREATE TABLE employees (id serial, first\_name varchar(50), last\_name varchar(50), details\_id integer REFERENCES employees\_details (id)); |



Связь, при которой у каждого сотрудника может быть только одна карточка с дополнительной информацией, называется “один к одному”. Для ее отображения применяется аннотация **@OneToOne**. Объявления связей «один к одному» и «один ко многим» почти аналогичны — за исключением использования аннотации, обозначающей вид связи.

Код сущностей будет выглядеть следующим образом:

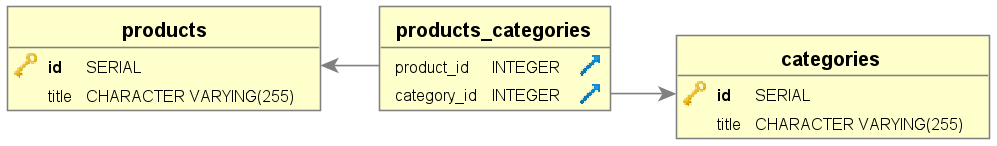
|  |
| --- |
| @Entity @Table(name = **"employees"**) **public** **class** Employee {    @Id    @GeneratedValue    @Column(name = **"id"**)    **private** Long id;     @Column(name = **"first\_name"**)    **private** String firstname;     @Column(name = **"last\_name"**)    **private** String lastname;          @OneToOne    @JoinColumn(name = **"details\_id"**)    **private** EmployeeDetails details;       *// ...*  }  @Entity @Table(name = **"employees\_details"**) **public** **class** EmployeeDetails {    @Id    @GeneratedValue    @Column(name = **"id"**)    Long id;     @OneToOne(mappedBy = **"details"**)    Employee employee;     *// ...*  } |

По details\_id в классе Employee мы получаем ссылку на “готовый” объект типа EmployeeDetails. В таблице employee\_details нет прямой ссылки на таблицу сотрудников, поэтому в классе EmployeeDetails мы используем mappedBy, чтобы получить ссылку на конкретного сотрудника.

Многие ко многим

А теперь посмотрим что изменится если мы захотим для каждому товару присваивать несколько категорий. В таком случае связь будет “многие ко многим” и нам понадобится промежуточная таблица, которая будет выступать связующим звеном между товарами и категориями.

|  |
| --- |
| DROP TABLE IF EXISTS categories CASCADE;  DROP TABLE IF EXISTS products CASCADE;  CREATE TABLE categories (id serial, title varchar(255), PRIMARY KEY (id));  CREATE TABLE products (id serial, title varchar(255), PRIMARY KEY (id));  CREATE TABLE products\_categories (product\_id integer REFERENCES products (id), category\_id integer REFERENCES categories (id)); |



Несмотря на кажущуюся сложность ее реализации в базе данных, в коде это выглядит довольно просто.

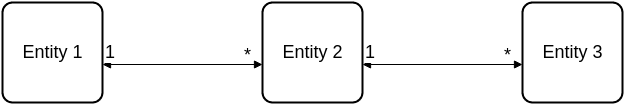
|  |
| --- |
| @Entity  @Table(name = **"products"**)  **public class** Product {     @Id     @GeneratedValue(strategy = GenerationType.***IDENTITY***)     @Column(name = **"id"**)  **private int id**;     @Column(name = **"title"**)  **private** String **title**;     @ManyToMany     @JoinTable(             name = **"products\_categories"**,             joinColumns = @JoinColumn(name = **"products\_id"**),             inverseJoinColumns = @JoinColumn(name = **"category\_id"**)     )  **private** List<Category> **categories**;  *// ...*  }  @Entity  @Table(name = **"categories"**)  **public class** Category {     @Id     @GeneratedValue(strategy = GenerationType.***IDENTITY***)     @Column(name = **"id"**)  **private int id**;     @Column(name = **"title"**)  **private** String **title**;     @ManyToMany     @JoinTable(             name = **"products\_categories"**,             joinColumns = @JoinColumn(name = **"category\_id"**),             inverseJoinColumns = @JoinColumn(name = **"products\_id"**)     )  **private** List<Product> **products**;  *// ...*  } |

Отображение связи «многие ко многим» осуществляется так:

* к атрибутам обоих классов применяется аннотация **@ManyToMany**;
* аннотация **@JoinTable** применяется для класса-владельца связи, в параметре **name** которой указывается имя таблицы соединения. В параметрах **joinColumns** и **inverseJoinColumns** указывается имя столбца, в котором отображается атрибут данного и ассоциированного класса соответственно;

**Доступ к атрибутам**

Представим, что есть сущность, полем которой является коллекция сущностей (связь «один ко многим»), а поле дочерней сущности — тоже коллекция сущностей и так далее. В итоге совокупность связей будет выглядеть следующим образом:



Вероятно, что при получении из БД **Entity 1** за ней тянулась бы целая цепочка коллекций других сущностей, и это сказалось бы на производительности негативно. Но JPA задает механизм для точного определения вида инициализации. Есть две стратегии выборки, которые задаются в атрибуте **fetch** аннотаций **@OneToOne**, **@OneToMany**, **@ManyToOne**, **@ManyToMany**:

* **LAZY** — «ленивая» выборка, при которой поля, являющиеся другими сущностями, не вытягиваются из базы данных вместе с основной сущностью, а запрашиваются отдельно (дополнительным запросом в БД), только при вызове геттера данного поля;
* **EAGER** — ранняя выборка, при которой поля, являющиеся сущностями, вытягиваются из базы вместе с основной сущностью с помощью одного запроса к БД.

Атрибут **fetch** аннотаций связи может иметь следующие значения:

* **FetchType.LAZY** — для ленивой выборки;
* **FetchType.EAGER** — для ранней выборки.

В коде стратегия выборки задается так:

|  |
| --- |
| @Entity @Table(name = **"country"**) **public** **class** Country {    @Id   @GeneratedValue(strategy = GenerationType.IDENTITY)   @Column(name = **"id"**)   **private** Long id;     @Column(name = **"title"**)   **private** String title;    @OneToMany(mappedBy = **"country"**, fetch = FetchType.EAGER)   **private** List<City> cities;     *// Геттеры и сеттеры* } |

По умолчания каждая из связей использует стратегии следующим образом:

|  |  |
| --- | --- |
| **Аннотация** | **Стратегия выборки по умолчанию** |
| @OneToOne | EAGER |
| @ManyToOne | EAGER |
| @OneToMany | LAZY |
| @ManyToMany | LAZY |

Для связи OneToOne можно применить только EAGER-стратегию.

Выбор стратегии остается на совести разработчика. EAGER-стратегия позволит загружать все данные с помощью небольшого количества запросов к БД. LAZY-стратегия не позволит заполнить всю используемую память. Вы будете контролировать, какой объект будет загружаться, но каждый раз придется осуществлять доступ к базе данных.

Задавать стратегию выборки можно и для простых атрибутов. Это можно осуществить с помощью аннотации **@Basic**.

**Каскадные операции**

Помимо стратегии выборки в атрибутах аннотаций связи можно указывать параметры каскадирования операций. Фактически, это означает, что при удалении сущности A из соответствующей таблицы в БД удаляется и ее дочерняя сущность B. Например, если есть сущность **Person**, которая включает в себя поле класса **Contact**, то вместе с объектом класса **Person** будут удалены строка в таблице **person** и ассоциированная с ней строка таблицы **contact**:

В коде каскадирование задается в атрибуте **cascade** аннотаций связи. Данный атрибут может принимать следующие значения перечисления **CascadeType** пакета **javax.persistence.CascadeType**:

* **CascadeType.ALL** — каскадирование будет применяться ко всем операциям;
* **CascadeType.REMOVE** — только к методу удаления;
* **CascadeType.PERSIST** — только к методу сохранения;
* **CascadeType.MERGE** — к методу обновления;
* **CascadeType.REFRESH** — к методу синхронизации с БД;
* **CascadeType.DETACH** — каскадирование применяется к методу удаления сущности из контекста постоянства (но не из БД).

В коде это выглядит так:

|  |
| --- |
| @ManyToOne(cascade = CascadeType.PERSIST) @JoinColumn(name = **"details\_id"**) **private** EmployeeDetails details; |

Кроме того, можно задавать несколько параметров:

|  |
| --- |
| @ManyToOne(cascade = { CascadeType.REMOVE, CascadeType.PERSIST }) @JoinColumn(name = **"details\_id"**) **private** EmployeeDetails details; |

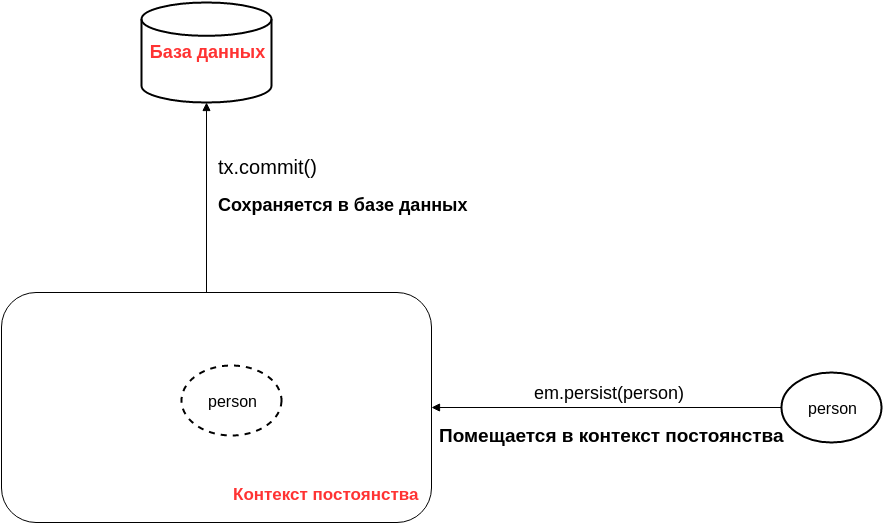
**Контекст постоянства и EntityManager**

Рассмотрим порядок взаимодействия наших сущностей с базой данных. Чтобы понять этот процесс, необходимо разобраться с понятием контекста постоянства. Этот контекст представляет собой набор сущностей, которые управляются Hibernate в данный момент. Все управление сущностями возлагается на менеджер сущностей — класс **EntityManager**.

Он обладает полным набором CRUD-операций. Данные операции вызываются следующими методами:

|  |
| --- |
| **public** **static** **void** main(String[] args) {     *// Получаем фабрику менеджеров сущностей*     EntityManagerFactory factory = **new** Configuration()             .configure(**"hibernate.cfg.xml"**)             .buildSessionFactory();     *// Из фабрики создаем EntityManager*     EntityManager em = factory.createEntityManager();      Person person = **new** Person(**"Ivan"**, **"Ivanov"**);      *// Открываем транзакцию*     em.getTransaction().begin();     *// Create (сохраняем в базе данных, и благодаря этому сущность*      *// становится управляемой Hibernate и заносится в контекст постоянства)*     em.persist(person);     *// Подтверждаем транзакцию*     em.getTransaction().commit();      em.getTransaction().begin();     *// Read (читаем сущность из базы данных по id)*     Person anotherPerson = em.find(Person.class, 1L);     em.getTransaction().commit();     anotherPerson.setFirstname(**"Artem"**);      em.getTransaction().begin();     *// Update*     em.merge(anotherPerson);     em.getTransaction().commit(); } |

Ниже графически представлен процесс сохранения в БД объекта-сущности **person**:



Данное изображение отражает всю суть работы контекста постоянства для любых операций. Главное, что следует запомнить, — **результат операции никак не отразится на базе данных, пока не будет произведена фиксация транзакции**.

Процесс, изображенный на данной схеме, состоит из следующих этапов:

* открытие транзакции;
* сохранение объекта в контексте постоянства с помощью метода **persist**;
* сохранение объекта в базе данных после фиксации транзакции.

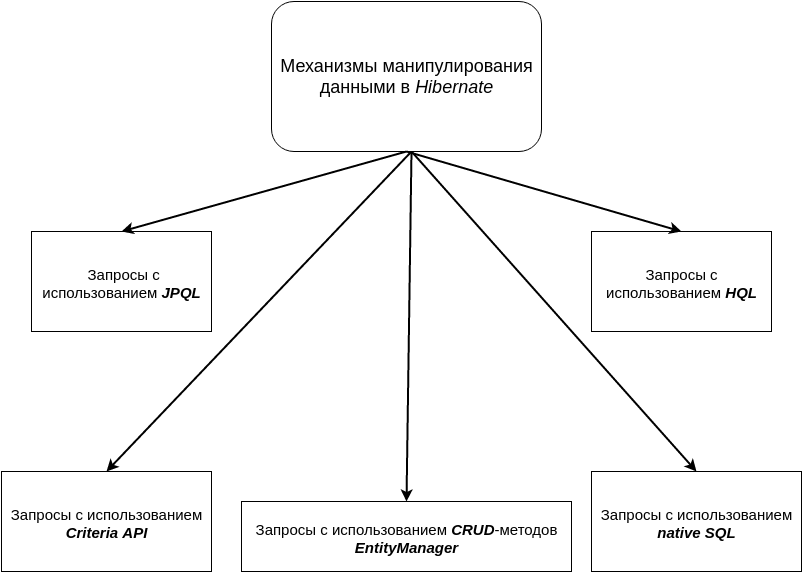
Данная последовательность характерна не только для метода **persist**, но и для остальных. В случае использования Hibernate совместно со Spring не будет необходимости самостоятельно открывать и закрывать транзакции — этим будет заниматься сам Spring.

**EntityManager** содержит и другие методы, предназначенные для работы с сущностями:

* **detach(Object obj)** — удаляет сущность **obj** из контекста постоянства (но не из БД), и объект перестает находиться под управлением Hibernate;
* **refresh(Object obj)** — синхронизирует сущность **obj** c БД. Ее поля будут иметь те значения, которые находились в столбцах строки БД на момент применения данного метода.

**JPQL**

Набор методов CRUD-класса **EntityManager** ограничивает возможности по манипулированию сущностями. Например, метод **find()** позволяет искать сущность только по идентификатору. Для создания более гибких запросов нужно использовать другие механизмы:



* **Запросы с использованием JPQL (Java Persistence Query Language)** — объектно-ориентированный язык запросов, который описан в спецификации JPA.  В отличие от SQL, он оперирует сущностями на уровне кода, а в дальнейшем поставщик постоянства транслирует эти запросы в SQL-запросы к БД;
* **Запросы с использованием HQL (Hibernate Query Language)** — аналогичен JPQL, но используется только в Hibernate;
* **Запросы с использованием CRUD-методов** — запросы к базе данных с помощью методов, рассмотренных в предыдущей главе;
* **Запросы с использованием Criteria API** — запросы, которые последовательно формируются с помощью объектов и методов.

Стоит отметить, что все языки запросов очень похожи на SQL.

Рассмотрим синтаксис языка запросов JPQL, который основан на HQL и позиционируется как более новая и стандартизованная версия этого языка. Запросы с использованием **Criteria API** оставим на самостоятельное изучение по дополнительным материалам.

**Обзор синтаксиса**

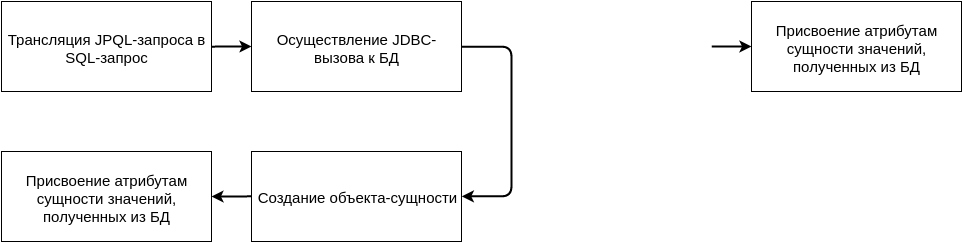
Главное отличие JPQL от SQL состоит в том, что JPQL-запросы манипулируют сущностями, то есть объектами классов. Самый простой JPQL-запрос, который делает выборку всех объектов-сущностей класса **Article** из базы данных, выглядит так:

|  |
| --- |
| SELECT a FROM Article a |

Особенности этого запроса:

* оператор **FROM** указывает на класс (в данном случае — класс **Article**), выборку объектов которого необходимо сделать из соответствующей ему таблицы в базе данных;
* в блоке оператора **FROM** указывается псевдоним класса (в данном случае — **a**).

При использовании JPQL-запросов происходит следующее:



Если необходимо применить определенный критерий поиска, то запрос будет выглядеть так:

|  |
| --- |
| SELECT a FROM Article a WHERE a.id = 2 |

В данном случае псевдоним **a** используется для доступа к атрибутам класса.

Возвращать можно не только объекты, но и атрибуты класса. Запрос, возвращающий атрибуты объекта, может выглядеть следующим образом:

|  |
| --- |
| SELECT a.firstname, a.lastname FROM Author a |

Если используется привязка параметров, запрос может быть таким:

|  |
| --- |
| SELECT a.firstname, a.lastname FROM Author a WHERE a.id = ?1 |

В случае именованных параметров:

|  |
| --- |
| SELECT a.firstname, a.lastname FROM Author a WHERE a.id = :id |

Можно указать поставщику постоянства, что необходимо создать объекты из возвращаемых из БД значений. Например:

|  |
| --- |
| SELECT NEW com.geekbrains.Person(a.firstname, a.lastname) FROM Author a |

Класс **Person** не обязан являться сущностью, но должен содержать конструктор с указанной в запросе сигнатурой.

**Запросы**

**Динамические запросы**

Изучим механизм, с помощью которого осуществляются запросы. Вся необходимая функциональность содержится в методах класса **EntityManager**.

Рассмотрим основные методы:

* **createQuery(String jpqlString)** — метод, принимающий строку JPQL-запроса и  возвращающий объект класса **Query**;
* **createNamedQuery(String name)** — метод для именованных запросов, принимающий их названия и возвращающий объекты класса **Query**;
* **createNativeQuery(String sqlString)** — метод для запросов с использованием SQL, возвращает объект класса **Query** и других.

Эти методы имеют свои перегруженные аналоги, принимающие дополнительный параметр типа **Class** или **Class<T>**, которые помогают избежать лишних преобразований типов.

Но стоит отметить, что все вышеперечисленные методы не обеспечивают выполнение запроса как такового — для этого необходимо использовать:

* **getSingleResult()** — для получения одиночного объекта в качестве конечного результата запроса;
* **getResultList()** — для получения коллекции объектов как конечного результата запроса;
* и другие.

Например, получение всех авторов может выглядеть следующим образом:

|  |
| --- |
| *// Осуществление запроса, возвращающего коллекцию* List<Author> authors = em.createQuery(**"SELECT a FROM Author a"**, Author.class).getResultList();  *// Осуществление запроса, возвращающего одиночный результат* Author author = em.createQuery(**"SELECT a FROM Author a WHERE a.id = 1"**, Author.class).getSingleResult(); |

**Именованные запросы**

Именованные запросы более производительны, чем динамические. Это связано с тем, что преобразование JPQL-запроса в SQL происходит сразу после запуска приложения. Чтобы выполнить именованный запрос, в классе, к которому он будет осуществляться, необходимо объявить аннотацию **@NamedQuery**, содержащую следующие атрибуты:

* **name** — название именованного запроса;
* **query** — JPQL-строка запроса.

Можно объявлять несколько именованных запросов с помощью множественной аннотации **@NamedQueries**.

Объявление именованных запросов:

|  |
| --- |
| @Entity @Table(name=**"author"**) @NamedQueries({   @NamedQuery(name = **"Author.findAll"**, query = **"SELECT a FROM Author a"**),   @NamedQuery(name = **"Author.findById"**, query = **"SELECT a FROM Author a WHERE a.id = :id"**) }) **public** **class** Author{   *// Fields, getter and setters* } |

В данном листинге показан пример объявления двух именованных запросов:

* **Author.findAll** для получения всех авторов;
* **Author.findById** (аналог метода **find** класса **EntityManager**), использующий именованные параметры.

Заметьте, что формат названий именованных запросов рекомендует употреблять имя класса в качестве префикса, а само название отражает операцию и критерий.

В коде использование именованных запросов будет выглядеть так:

|  |
| --- |
| List<Author> authors = em.createNamedQuery(**"Author.findAll"**, Author.class).getResultList(); Author author = em.createNamedQuery(**"Author.findById"**, Author.class).setParameter(**"id"**, 1).getSingleResult(); |

**Аннотации**

Давайте рассмотрим список аннотаций, применяемые в Hibernate.

1. Каждый класс хранимой сущности должен иметь аннотацию **@Entity**;

2. Аннотации **@Table** позволяет задать имя таблицы, в которую будут отображаться объекты данной сущности, и настроить индексы;

|  |
| --- |
| **@Table(name = "demo\_annotated", indexes = {**  **@Index(name = "name\_idx", columnList = "name"),**  **@Index(name = "id\_name\_idx", columnList = "id, name"),**  **@Index(name = "unique\_name\_idx", columnList = "name", unique = true)**  **})** |

3. Аннотации **@OneToOne**, **@OneToMany**, **@ManyToOne**, **@ManyToMany** обозначают связи между сущностями;

4. **@Column** отвечает за настройки столбца в таблице. С помощью параметра **name** указывается имя столбца в которое будет записано значение размеченного поля. Для “ручного” формирования запроса с помощью которого будет построен столбец можно воспользоваться параметром **columnDefinition**. Здесь же можно указать ограничение NOT NULL (**nullable** = false). Чтобы запретить изменение значение какого-либо столбца параметр **updatable** переводится в false.

|  |
| --- |
| @Column(name = **"manual\_def\_str"**, columnDefinition = **"VARCHAR(50) NOT NULL UNIQUE CHECK (NOT substring(lower(manual\_def\_str), 0, 5) = 'admin')"**)  String **manualDefinedString**;  @Column(name = **"short\_str"**, nullable = **false**, length = 10) *// varchar(10)*  String **shortString**;  @Column(name = **"created\_at"**, updatable = **false**)  LocalDateTime **createdAt**; |

5. Для автогенерации времени создания объекта в базе данных и времени его обновления используются аннотации **@CreationTimestamp** и **@UpdateTimestamp**;

6. При реализации связи **@ManyToOne**, для создания внешнего ключа указывается следующая форма аннотации **@JoinColumn**

|  |
| --- |
| @ManyToOne  @JoinColumn(         name = **"product\_id"**,         nullable = **false**,         foreignKey = @ForeignKey(name = **"FK\_PRODUCT\_ID"**)  )  Product **product**; |

6. Любой класс хранимой сущности обязан иметь идентифицирующий атрибут - поле, помеченное аннотацией **@Id**. Аннотация **@GeneratedValue** по умолчанию указывает что генерация будет выполнятся автоматически, и позволяет указать способ генерации.

|  |
| --- |
| **public class** Product {     @Id     @GeneratedValue     @Column(name = **"id"**)     Long **id**; |

8. **@Version** поле используется для версионирования, о котором речь пойдет ниже.

**Оптимистическое управление параллельным доступом**

Оптимистическое управление параллельным доступом подходит для случаев, когда изменения вносятся редко и в рамках одной транзакции допустимо позднее обнаружение конфликтов. Для оптимистического управления необходимо включить версионирование. При таком подходе будет “побеждать” первая подтвержденная транзакция.

Для добавления версионирования достаточно добавить в классы, помеченные аннотацией @Entity поле с аннотацией @Version.

|  |
| --- |
| @Entity @Table(name = **"items"**) **public** **class** Item {    @Id    @GeneratedValue    @Column(name = **"id"**)    Long id;     @Column(name = **"val"**)    **int** val;     @Column(name = **"junkField"**)    @OptimisticLock(excluded = **true**)    **int** junkField;     @Version    **long** version;     **public** **void** setVal(**int** val) {        **this**.val = val;    }     **public** **long** getVersion() {        **return** version;    }     **public** Item() {    }     **public** Item(**int** val) {        **this**.val = val;    }     @Override    **public** String toString() {        **return** String.format(**"Item [ id = %d, val = %d, version = %d ]"**, id, val, version);    } } |

В таком случае, каждому экземпляру данной сущности будет присваиваться версия, которая отображается на отдельный столбец в таблицы базы данных. Для поля version можно добавить геттер, но ни в коем случае не должно быть сеттера, так как изменением этого поля занимается сам Hibernate. По сути, версия - это просто счетчик. Давайте посмотрим на следующий пример:

|  |
| --- |
| *// ... тут стандартный запуск SessionFactory* session = factory.getCurrentSession();  session.beginTransaction();  Item item = session.find(Item.**class**, 1L);  System.***out***.println(item.getVersion()); *// <- 1*  item.setVal(20);  session.flush();  System.***out***.println(item.getVersion()); *// <- 2*  item.setVal(30);  session.flush();  System.***out***.println(item.getVersion()); *// <- 3*  session.getTransaction().rollback();  session = factory.getCurrentSession();  session.beginTransaction();  item = session.find(Item.**class**, 1L);  System.***out***.println(item.getVersion()); *// <- 1*  session.getTransaction().commit();  session.close();  *// … а тут завершение работы* |

При внесении изменений в состояние item, Hibernate накапливает эти изменения, но не посылает запросы в базу данных. Выполнение session.flush() выталкивает контекст хранения, выполняя запросы в БД. В результате, после каждого flush() версия растет. Если выполняется rollback(), то само собой изменения версии не подтверждаются.

Для проверки работы такого варианта оптимистической блокировки, можно воспользоваться следующим кодом.

|  |
| --- |
| *// ...* **new** Thread(() -> {    System.out.println(**"Thread #1 started"**);    Session session = factory.getCurrentSession();    session.beginTransaction();    Item item = session.get(Item.class, 1L); // <- version = 1    item.setVal(100);    uncheckableSleep(1000);    session.save(item);    session.getTransaction().commit(); // version увеличивается до 2    System.out.println(**"Thread #1 committed"**);    **if** (session != **null**) {        session.close();    }    countDownLatch.countDown(); }).start();  **new** Thread(() -> {    System.out.println(**"Thread #2 started"**);    Session session = factory.getCurrentSession();    session.beginTransaction();    Item item = session.get(Item.class, 1L); // <- version = 1    item.setVal(200);    uncheckableSleep(3000);    **try** {        session.save(item);        session.getTransaction().commit(); // в момент подтверждения транзакции во втором потоке производится сравнение версии при старте транзакции (1) и текущим значением версии (2)        System.out.println(**"Thread #2 committed"**);    } **catch** (OptimisticLockException e) {        session.getTransaction().rollback();        System.out.println(**"Thread #2 rollback"**);        e.printStackTrace();    }    **if** (session != **null**) {        session.close();    }    countDownLatch.countDown(); }).start(); **try** {    countDownLatch.await(); } **catch** (InterruptedException e) {    e.printStackTrace(); } *// ...* |

*\* unckeckableSleep(int ms) метод, выполняющий Thread.sleep(), с перехватом InterruptedException.*  
Два потока параллельно пытаются изменить состояние item. При старте, в каждой транзакции выполняется “запоминание версии объекта”, которая равна 1. Транзакция может быть успешно завершена только в том случае, если версия объекта на момент начала и подтверждения транзакции совпадают.

Из кода видно, что первый поток подтвердит транзакцию раньше второго, при этом будет произведено сравнение версий, и поскольку **1 == 1**, транзакция удачно завершится и версия увеличится на 1. Через пару секунд, второй поток попытается также завершить транзакцию, но версии будут отличаться **1 != 2**, и в этом случае будет сгенерировано исключение OptimisticLockException, после перехвата которого выполняется rollback().

Если изменение какого-либо поля не должно влиять на версию объекта, то такое поле можно пометить как @OptimisticLock(excluded = **true**), в примере кода из начала пункта, такой аннотацией было помечено поле int junkField.

**Пессимистические блокировки**

Давайте рассмотрим случай пессимистической блокировки.

|  |
| --- |
| // ...  session = factory.getCurrentSession();  session.beginTransaction();  int sumValue = **0**;  List<Item> items = session.createQuery(**"SELECT i FROM Item i;"**, Item.**class**)  .setLockMode(LockModeType.***PESSIMISTIC\_READ***)         .getResultList();  **for** (Item o : items) {     sumValue += o.**getVal()**;  }  session.getTransaction().commit();  // ... |

Допустим мы хотим просуммировать значения всех item’ов в нашей таблице, и сделать это надо именно на стороне нашего приложения, а не базы данных. В таком случае, при получении списка объектов из базы данных, мы не можем давать другим транзакциям изменять значения этих элементов. Для этого может быть установлена пессимистическая блокировка с помощью метода setLockMode(). При выборе в качестве аргумента LockModeType.PESSIMISTIC\_READ, полученные записи будут доступны другим транзакциям только для чтения, такой режим аналогичен блокировке PosetgreSQL “FOR SHARE”. Если выбрать LockModeType.PESSIMISTIC\_WRITE, то строки заблокируются как для чтения, так и для записи, что в PostgreSQL аналогично “FOR UPDATE”. Блокировки будут сняты только по завершению текущей транзакции.

**Практическое задание**

1. В базе данных необходимо реализовать возможность хранить информацию о покупателях (id, имя) и товарах (id, название, стоимость). У каждого покупателя свой набор купленных товаров.

Задача: написать тестовое консольное приложение, которое позволит посмотреть, какие товары покупал клиент, какие клиенты купили определенный товар, и предоставит возможность удалять из базы товары/покупателей.

1. \*\* Добавить детализацию по паре «покупатель — товар»: сколько стоил товар в момент покупки клиентом.

**Дополнительные материалы**

1. <https://hibernate.org/orm/documentation/5.4/>
2. **GEEKCHANGE: "Java. Изучаем Hibernate ORM для работы с базами данных"** <https://www.youtube.com/watch?v=emg94BI2Jao> (Введение в Hibernate)

**Используемая литература**

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Крис Шефер, Кларенс Хо. Spring 4 для профессионалов (4-е издание).
2. Крейг Уоллс. Spring в действии.

**Доступ к данным в Spring. Часть 2**

DAO. Spring Data JPA. Сервис-уровень.

**Оглавление**

[Доступ к базе данных из Spring приложения](https://docs.google.com/document/d/1nIsVBLF24ZVTpA51jeLTUowc-OfGnZ6G3uZ1IixHK_0/edit#heading=h.2imvmz22iddp)

[DAO](https://docs.google.com/document/d/1nIsVBLF24ZVTpA51jeLTUowc-OfGnZ6G3uZ1IixHK_0/edit#heading=h.uu765c4qkrwj)

[Spring Data JPA](https://docs.google.com/document/d/1nIsVBLF24ZVTpA51jeLTUowc-OfGnZ6G3uZ1IixHK_0/edit#heading=h.35nkun2)

[Транзакции и уровень сервисов](https://docs.google.com/document/d/1nIsVBLF24ZVTpA51jeLTUowc-OfGnZ6G3uZ1IixHK_0/edit#heading=h.1ksv4uv)

[Практика](https://docs.google.com/document/d/1nIsVBLF24ZVTpA51jeLTUowc-OfGnZ6G3uZ1IixHK_0/edit#heading=h.af5ipntmeejo)

[Добавление зависимостей](https://docs.google.com/document/d/1nIsVBLF24ZVTpA51jeLTUowc-OfGnZ6G3uZ1IixHK_0/edit#heading=h.2jxsxqh)

[Конфигурация](https://docs.google.com/document/d/1nIsVBLF24ZVTpA51jeLTUowc-OfGnZ6G3uZ1IixHK_0/edit#heading=h.z337ya)

[Создание репозиториев](https://docs.google.com/document/d/1nIsVBLF24ZVTpA51jeLTUowc-OfGnZ6G3uZ1IixHK_0/edit#heading=h.3j2qqm3)

[Создание сервис-уровня](https://docs.google.com/document/d/1nIsVBLF24ZVTpA51jeLTUowc-OfGnZ6G3uZ1IixHK_0/edit#heading=h.1y810tw)

[Практическое задание](https://docs.google.com/document/d/1nIsVBLF24ZVTpA51jeLTUowc-OfGnZ6G3uZ1IixHK_0/edit#heading=h.4i7ojhp)

[Дополнительные материалы](https://docs.google.com/document/d/1nIsVBLF24ZVTpA51jeLTUowc-OfGnZ6G3uZ1IixHK_0/edit#heading=h.1ci93xb)

[Используемая литература](https://docs.google.com/document/d/1nIsVBLF24ZVTpA51jeLTUowc-OfGnZ6G3uZ1IixHK_0/edit#heading=h.3whwml4)

**Доступ к базе данных из Spring приложения**

На прошлом занятии мы рассмотрели принципы работы с Hibernate, теперь пришло время применить это на практике при разработке веб-приложения.

**DAO**

**DAO (Data Access Object)** — это уровень доступа к данным. Вся функциональность DAO основывается на классе **EntityManager**. Чтобы создать DAO-уровень для сущности, необходимо выполнить следующие действия:

* создать отдельный пакет для классов уровня доступа к данным, например **com.geekbrains.dao**;
* создать интерфейс доступа к сущности, например **ArticleDAO**;
* в интерфейсе объявить методы, исходя из набора требующихся операций над сущностью;
* создать класс, имплементирующий данный интерфейс, и реализовать в нем методы интерфейса, используя **EntityManager** для обеспечения  функциональности.

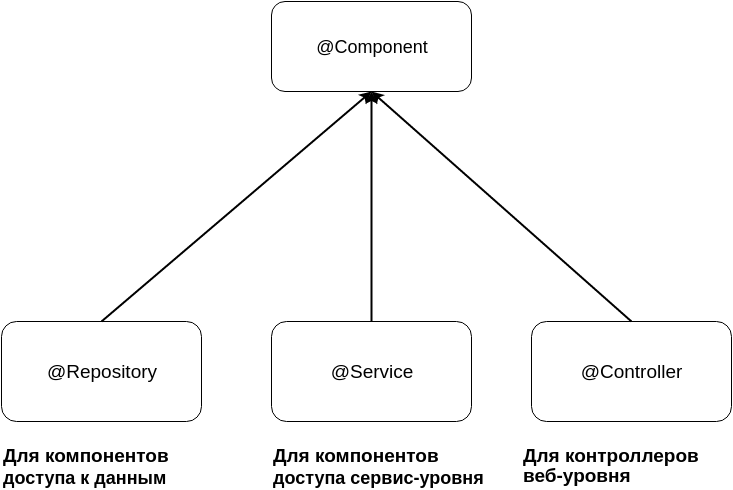
Предположим, что для сущности **Article** необходимо реализовать следующие операции: поиск всех сущностей, сохранение сущности **Article**, получение сущности по id, обновление сущности, удаление сущности по id. Исходя из этого, интерфейс доступа будет выглядеть так:

|  |
| --- |
| public interface ArticleDAO {    List<Article> findAll();    void save(Article article);    Article findById(Long id);    void update(Article article);    void delete(Article article);  } |

В приведенном интерфейсе нет специальных аннотаций. Реализация данного интерфейса:

|  |
| --- |
| @Repository  public class ArticleDAOImpl implements ArticleDAO {    @PersistenceContext    private EntityManager entityManager;    @Override    public List<Article> findAll() {      return entityManager.createQuery("from Article", Article.class).getResultList();    }    @Override    public void save(Article article) {      entityManager.persist(article);    }    @Override    public Article findById(Long id) {      return entityManager.find(Article.class, id);    }    @Override    public void update(Article article) {      entityManager.merge(article);    }    @Override    public void delete(Article article) {      entityManager.remove(article);    }  } |

Приведенный выше класс также является компонентом Spring, но он помечен аннотацией **@Repository**. Это уточняющая аннотация по отношению к **@Component**. Указывает на то, что данный компонент необходим для доступа к данным. Фактически, **@Repository** является одним из видов аннотации **@Component**. Данный класс также будет управляться контейнером Spring и будет пригодным для внедрения в другие классы и компоненты.



В данном коде происходит внедрение **EntityManager** c помощью аннотации **@PersistenceContext**, а не **@Autowired**. Это связано с тем, что в проекте могут использоваться сразу несколько источников данных, а аннотация **@PersistenceContext** обладает широким набором атрибутов, которые необходимы для точного указания настроек контекста постоянства.

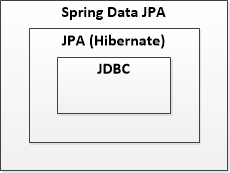
**Spring Data JPA**

Долгое время для организации доступа к данным использовался подход, описанный выше. Но он состоит в основном из тривиальных задач: реализовать интерфейс, его имплементацию, внедрить менеджер сущностей и подобных. При этом от сущности к сущности код базовых операций (сохранения, чтения, удаления и обновления данных) не особо отличается, если в коде выше, в ArticleDAOImpl заменить слово/класс Article на любой другой класс (например, Product), то все будет работать. Давайте проверим:

|  |
| --- |
| public interface ProductDAO {    List<Product> findAll();    void save(Product product);    Product findById(Long id);    void update(Product product);    void delete(Product product);  }  @Repository  public class ProductDAOImpl implements ProductDAO {    @PersistenceContext    private EntityManager entityManager;    @Override    public List<Product> findAll() {      return entityManager.createQuery("from Product", Product.class).getResultList();    }    @Override    public void save(Product product) {      entityManager.persist(product);    }    @Override    public Product findById(Long id) {      return entityManager.find(Product.class, id);    }    @Override    public void update(Product product) {      entityManager.merge(product);    }    @Override    public void delete(Product product) {      entityManager.remove(product);    }  } |

Как видите каких-то изменений в коде кроме замены слова Article/article -> Product/product никаких больше изменений не произошло. Если мы возьмем 100 таких классов, то получим столько же очень похожих кусков кода. Чтобы избавить от написания одного и того же кода, фреймворк Spring предоставляет модуль Spring Data JPA.

Spring Data JPA, фактически, является очередной абстракцией над Hibernate (Hibernate же является реализацией спецификации JPA). Ну а на самом нижнем уровне по-прежнему JDBC.



Что нам дает и зачем нужен еще один уровень абстракции? Spring Data генерирует стандартный код для работы с базами данных, для этого достаточно указать сущность для которой нужно сгенерировать этот код и тип ID этой сущности. Например, если мы создадим интерфейс и унаследуем его от CrudRepository:

|  |
| --- |
| @Repository  public interface ArticleRepository extends CrudRepository<Article, Long> {  } |

То нам будет “из коробки” доступен набор CRUD-операций по работе с сущностью Article: save(...), saveAll(...), findAll(), findById(...), count(), delete(...), deleteById(...), deleteAll(), deleteAll(...) и т.д. Из названий методов должно быть понятно что они позволяют делать. То есть мы можем заинжектить подобный репозиторий в сервис и спокойно с ним работать, весь код, реализующий эти методы подготовит сам Spring.

Spring Data JPA предоставляет следующий набор репозиториев. Каждый следующий расширяет возможности предыдущего.

1. **Базовый интерфейс**

|  |
| --- |
| public interface Repository<T, ID> {  } |

1. **CrudRepository** - предоставляет набор базовых CRUD операций

|  |
| --- |
| public interface CrudRepository<T, ID> extends Repository<T, ID> {    <S extends T> S save(S var1);    <S extends T> Iterable<S> saveAll(Iterable<S> var1);    Optional<T> findById(ID var1);    boolean existsById(ID var1);    Iterable<T> findAll();    Iterable<T> findAllById(Iterable<ID> var1);    long count();    void deleteById(ID var1);    void delete(T var1);    void deleteAll(Iterable<? extends T> var1);    void deleteAll();  } |

1. **PagingAndSortingRepository** - добавляет возможность использования пагинации и сортировки

|  |
| --- |
| public interface PagingAndSortingRepository<T, ID extends Serializable> extends CrudRepository<T, ID> {    Iterable<T> findAll(Sort sort);    Page<T> findAll(Pageable pageable);  } |

1. **JpaRepository**

|  |
| --- |
| public interface JpaRepository<T, ID> extends PagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T> {    List<T> findAll();    List<T> findAll(Sort var1);    List<T> findAllById(Iterable<ID> var1);    <S extends T> List<S> saveAll(Iterable<S> var1);    void flush();    <S extends T> S saveAndFlush(S var1);    void deleteInBatch(Iterable<T> var1);    void deleteAllInBatch();    T getOne(ID var1);    <S extends T> List<S> findAll(Example<S> var1);    <S extends T> List<S> findAll(Example<S> var1, Sort var2);  } |

В таблице ниже приведено назначение стандартных методов:

|  |  |
| --- | --- |
| **Метод** | **Назначение** |
| long count() | Возвращает количество доступных сущностей |
| void delete(T entity) | Удаляет указанную сущность |
| void deleteAll() | Удаляет все сущности |
| void deleteAll(Iterable<? extends T> entities) | Удаляет указанный набор сущностей |
| void deleteById(ID id) | Удаляет сущность с указанным id |
| boolean existsById(ID id) | Проверяет существование сущности с указанным id |
| Iterable<T> findAll() | Возвращает все объекты данного типа |
| Iterable<T> findAllById(Iterable<ID> ids) | Получает все объекты по набору id |
| Optional<T> findById(ID id) | Возвращает сущность по id |
| <S extends T> S save(S entity) | Сохраняет указанную сущность |
| <S extends T> Iterable<S> saveAll(Iterable<S> entities) | Сохраняет указанный набор сущностей |

Помимо методов, предоставляемых репозиториями Spring Data, мы можем создавать свои собственные. Функционал методов будет определяться их названием. Если создать метод **Author findByTitle(String title)**, то мы сможем искать статьи по названию; **Author findByTitleAndAuthor** - по названию и автору; **List<Author> findAllByAuthor(String author)** - все статьи по указанному автору. То есть Spring Data разбирает имя метода на части и по нему формирует запрос.

|  |
| --- |
| @Repository  public interface ArticleRepository extends JpaRepository<Article, Long> {    Article findByTitle(String title);  } |

В коде определен собственный метод поиска сущности по заданному критерию. Spring Data преобразовывает название метода и его сигнатуру в соответствующий запрос. В таблице ниже приведены основные конструкции, которые могут быть использованы в названии методов репозитория.

|  |  |  |
| --- | --- | --- |
| **Ключевое слово** | **Пример имени метода** | **JPQL код** |
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Is, Equals | findByFirstname,  findByFirstnameIs,  findByFirstnameEquals | … where x.firstname = ?1 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age <= ?1 |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where x.age >= ?1 |
| After | findByStartDateAfter | … where x.startDate > ?1 |
| Before | findByStartDateBefore | … where x.startDate < ?1 |
| IsNull | findByAgeIsNull | … where x.age is null |
| IsNotNull,NotNull | findByAge(Is)NotNull | … where x.age not null |
| Like | findByFirstnameLike | … where x.firstname like ?1 |
| NotLike | findByFirstnameNotLike | … where x.firstname not like ?1 |
| StartingWith | findByFirstnameStartingWith | … where x.firstname like ?1(parameter bound with appended %) |
| EndingWith | findByFirstnameEndingWith | … where x.firstname like ?1(parameter bound with prepended %) |
| Containing | findByFirstnameContaining | … where x.firstname like ?1(parameter bound wrapped in %) |
| OrderBy | findByAgeOrderByLastnameDesc | … where x.age = ?1 order by x.lastname desc |
| Not | findByLastnameNot | … where x.lastname <> ?1 |
| In | findByAgeIn(Collection<Age> ages) | … where x.age in ?1 |
| NotIn | findByAgeNotIn(Collection<Age> ages) | … where x.age not in ?1 |
| True | findByActiveTrue() | … where x.active = true |
| False | findByActiveFalse() | … where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | … where UPPER(x.firstame) = UPPER(?1) |

Разберем несколько примеров из таблицы выше:

* **findByFirstname(String firstname)** - поиск пользователя с указанным именем;
* **findByLastnameAndFirstname(String lastname, String firstname)** - поиск пользователя с указанными фамилией и именем;
* **findByAgeIsNull()** - поиск пользователей, у которых поле age не заполнено;

Если функциональности методов **JpaRepository** недостаточно, а описать запрос через название метода проблематично, то можно воспользоваться JPQL:

|  |
| --- |
| @Query("select a from Article a where a.author = :author")  List<Article> findArticleByAuthor(@Param("author") Author author); |

Этих объявлений вполне хватит, чтобы Spring самостоятельно создал объект класса, реализующего этот интерфейс. Чтобы использовать этот класс, необходимо произвести внедрение с помощью **@Autowired** по данному интерфейсу. О конфигурировании проекта для использования Spring Data JPA поговорим в разделе «Практика».

**Транзакции и уровень сервисов**

Между уровнем доступа к данным и веб-уровнем, который будет рассмотрен на следующем уроке, располагается уровень сервисов. В сервис-уровень помещена необходимая бизнес-логика, которая оперирует данными, получаемыми из уровня доступа к данным и веб-уровня. В методах сервис-уровня происходит работа с транзакциями.

Управление транзакциями бывает двух видов:

* управление приложением — явное открытие и фиксация транзакций разработчиком;
* управление контейнером — управление транзакциями делегируется контейнеру, в котором выполняется приложение.

По определению, открытие и фиксация транзакции происходит на сервис-уровне. Ведь на нем выполняется бизнес-логика, которая может оперировать несколькими сущностями, а значит, несколькими классами уровня доступа к данным. Но каким образом организовать транзакции на сервис-уровне, если наш **EntityManager** инкапсулирован в классах DAO-уровня и поэтому вызывать в сервис-уровне метод **em.getTransaction()** невозможно? В данном случае необходимо делегировать управление транзакциями контейнеру.

Чтобы указать контейнеру, что в методе необходимо открыть транзакцию и зафиксировать ее по окончании выполнения метода, нужно использовать аннотацию **@Transactional**:

|  |
| --- |
| @Service  public class ArticleServiceImpl implements ArticleService {    @Autowired    private ArticleRepository articleRepository;      public List<Article> getAll() {      return articleRepository.findAll();    }    @Override    @Transactional(readOnly = true)    public List<Article> getAll() {      return articleRepository.findAll();    }    @Override    @Transactional(readOnly = true)    public Article get(Long id) {      return articleRepository.findOne(id);    }    @Override    @Transactional    public void save(Article article) {      articleRepository.save(article);    }  } |

Здесь ко всем методам сервиса применяется аннотация **@Transactional**. Она указывает контейнеру, что необходимо открыть транзакции перед началом выполнения кода метода и закрыть их после того, как весь код метода выполнен. Если транзакция подразумевает только чтение из БД, то можно воспользоваться атрибутом **readOnly** и указать значение **true**.

**Практика**

В данном разделе рассмотрим пример добавления репозитория и сервис-уровня.

**Добавление зависимостей**

Так как в проекте будет использоваться **Spring Data JPA**, необходимо добавить зависимость:

|  |
| --- |
| <dependency>      <groupId>org.springframework.data</groupId>      <artifactId>spring-data-jpa</artifactId>      <version>2.0.9.RELEASE</version>  </dependency> |

**Конфигурация**

Класс конфигурации будет выглядеть так:

|  |
| --- |
| @Configuration  @EnableJpaRepositories("com.geekbrains.app.repositories")  @EnableTransactionManagement  @ComponentScan("com.geekbrains")  public class AppConfig {    @Bean(name="dataSource")    public DataSource getDataSource() {      // Создаем источник данных      DriverManagerDataSource dataSource = new DriverManagerDataSource();      // Задаем параметры подключения к базе данных      dataSource.setUrl("jdbc:mysql://localhost:3306/geekbrains-lesson3");      dataSource.setUsername("geek");      dataSource.setDriverClassName("com.mysql.jdbc.Driver");      dataSource.setPassword("geek");      return dataSource;    }      @Bean(name="entityManagerFactory")    public LocalContainerEntityManagerFactoryBean getEntityManager() {      // Создаем класса фабрики, реализующей интерфейс      // FactoryBean<EntityManagerFactory>      LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();      // Задаем источник подключения      factory.setDataSource(getDataSource());      // Задаем адаптер для конкретной реализации JPA,      // указывает, какая именно библиотека будет использоваться в качестве      // поставщика постоянства      factory.setJpaVendorAdapter(new HibernateJpaVendorAdapter());        // Указание пакета, в котором будут находиться классы-сущности      factory.setPackagesToScan("com.geekbrains");        // Создание свойств для настройки Hibernate      Properties jpaProperties = new Properties();      // Указание диалекта конкретной базы данных    jpaProperties.put("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");      // Указание максимальной глубины связи      jpaProperties.put("hibernate.max\_fetch\_depth", 3);        // Максимальное количество строк, возвращаемых за один запрос из БД      jpaProperties.put("hibernate.jdbc.fetch\_size", 50);        // Максимальное количество запросов при использовании пакетных операций      jpaProperties.put("hibernate.jdbc.batch\_size", 10);        // Включает логирование      jpaProperties.put("hibernate.show\_sql", true);        factory.setJpaProperties(jpaProperties);      return factory;    }      @Bean(name = "transactionManager")    public JpaTransactionManager transactionManager(EntityManagerFactory  entityManagerFactory) {      // Создание менеджера транзакций      JpaTransactionManager tm = new JpaTransactionManager();      tm.setEntityManagerFactory(entityManagerFactory);      return tm;    }  } |

В данном коде появились следующие элементы:

* аннотация **@EnableJpaRepositories** — обеспечивает возможность использования Spring Data JPA. В качестве параметра указывается пакет, в котором будут находиться классы-репозитории;
* аннотация **@EnableTransactionManagement** — указывает Spring на необходимость в управлении транзакциями;
* бин **transactionManager** — менеджер транзакций, который работает «поверх» менеджера сущностей.

**Создание репозиториев**

Перед разработкой интерфейсов создадим пакет, в котором будут находиться все интерфейсы репозиториев (например, com.geekbrains.app.repositories). И добавим интерфейсы репозиториев, которые будут расширять интерфейс **JpaRepository**.

Для сущностей класса **Author**:

|  |
| --- |
| @Repository  public interface AuthorsRepository extends JpaRepository<Author, Long> {  } |

Он расширяет интерфейс JpaRepository, которому необходимо указать два параметра для типизации (класс сущности и класс поля id сущности). По необходимости в данный интерфейс можно добавить собственные методы, используя предыдущий раздел о Spring Data JPA и дополнительный материал № 1.

Для сущностей класса **Article**:

|  |
| --- |
| @Repository  public interface ArticlesRepository extends JpaRepository<Article, Long> {  } |

**Создание сервис-уровня**

Для уровня сервисов необходимо создать пакет, в котором будут находиться классы-сервисы, — com.geekbrains.app.services.

Сервис-уровень создается в два этапа:

* создание интерфейсов;
* создание классов, реализующих данные интерфейсы.

Необходимо разработать следующие интерфейсы:

* **ArticleService**;
* **AuthorService**;

Код интерфейса **ArticleService**:

|  |
| --- |
| public interface ArticleService {    List<Article> getAll();    Article get(Long id);    void save(Article article);  } |

Код интерфейса **AuthorService**:

|  |
| --- |
| public interface AuthorService {    Author get(Long id);    List<Author> getAll();    void save(Author author);    void remove(Author author);  } |

Теперь разработаем классы, реализующие данные интерфейсы:

* **ArticleServiceImpl**;
* **AuthorServiceImpl**;

Они будут являться компонентами Spring.

Код класса **ArticleServiceImpl**:

|  |
| --- |
| @Service  public class ArticleServiceImpl implements ArticleService {    private ArticlesRepository articlesRepository;      @Autowired    public void setArticlesRepository(ArticlesRepository articlesRepository) {      this.articlesRepository = articlesRepository;    }      @Override    @Transactional(readOnly = true)    public List<Article> getAll() {      return articlesRepository.findAll();    }    @Override    @Transactional(readOnly = true)    public Article get(Long id) {      return articlesRepository.findOne(id);    }    @Override    @Transactional    public void save(Article article) {      articlesRepository.save(article);    }  } |

Класс **AuthorServiceImpl**:

|  |
| --- |
| @Service  public class AuthorServiceImpl implements AuthorService {    private AuthorsRepository authorsRepository;      @Autowired    public void setAuthorsRepository(AuthorsRepository authorsRepository) {      this.authorsRepository = authorsRepository;    }      @Override    @Transactional(readOnly = true)    public Author get(Long id) {      return authorsRepository.findOne(id);    }    @Override    @Transactional(readOnly = true)    public List<Author> getAll() {      return authorsRepository.findAll();    }    @Override    @Transactional    public void save(Author author) {      authorsRepository.save(author);    }    @Override    @Transactional    public void remove(Author author) {      authorsRepository.delete(author);    }  } |

**Практическое задание**

1. Создать сущность «товар» (id, название, стоимость) и соответствующую таблицу в БД. Заполнить таблицу тестовыми данными (20 записей).
2. Сделать страницу, в которую будут выведены эти записи.
3. С помощью GET-запроса указывать фильтрацию по:
   1. только минимальной (т.е. ищем товары у которых цена выше min),
   2. только максимальной (ищем товары, цена которых ниже max),
   3. или минимальной и максимальной цене (ищем товары, цена которых укладывается в пределы от min до max).
   4. \* **min** и **max** это параметры запроса (.../show?**min**=10&**max**=100 или .../show?**min**=1000)
4. \* Добавить постраничное отображение (по 5 записей на странице).

**Дополнительные материалы**

1. [Создание собственных методов репозитория](http://docs.spring.io/spring-data/jpa/docs/current/reference/html/#project) (п. 5.3.2).
2. Крис Шефер, Кларенс Хо. Spring 4 для профессионалов (4-е издание) — стр. 425–450.

**Используемая литература**

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Крис Шефер, Кларенс Хо. Spring 4 для профессионалов (4-е издание).
2. Крейг Уоллс. Spring в действии.

**Thymeleaf**

Шаблонизатор Thymeleaf. Интеграция с фреймворком Spring. Выражения. Операторы.

**Оглавление**

[Thymeleaf](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.2et92p0)

[Типы используемых шаблонов](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.tyjcwt)

[Диалекты Thymeleaf](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.3dy6vkm)

[Интеграция со Spring](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.1t3h5sf)

[Вариация Spring Boot](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.4d34og8)

[Отображение строк из файлов message.properties и интернационализация](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.i89ka4gwnsfd)

[Отображение атрибутов модели](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.17dp8vu)

[Отображение атрибутов коллекций](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.3rdcrjn)

[Форматированный вывод](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.6ezdppmkea5)

[Обработка форм](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.lnxbz9)

[Валидация полей формы](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.35nkun2)

[Упрощение стилизации CSS на основе ошибок](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.1ksv4uv)

[Условные выражения](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.hve68u5gllom)

[Практическое задание](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.2jxsxqh)

[Дополнительные материалы](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.3j2qqm3)

[Используемая литература](https://docs.google.com/document/d/1xGNGHTzshaSrJDIL9yD8LVIl59YJ51Mx_9mnOs9_C-w/edit#heading=h.1y810tw)

# Thymeleaf

Thymeleaf — это современный серверный движок Java-шаблонов для веб- и автономных сред, способный обрабатывать HTML, XML, JavaScript, CSS и простой текст. Основная цель Thymeleaf — предоставить элегантный и удобный способ создания шаблонов. Он основывается на концепции Natural Templates, внедряя свою логику в файлы шаблона таким образом, чтобы он не влиял на использование прототипа дизайна. Использование Thymeleaf улучшает дизайн и способствует более тесному взаимодействию между группами backend- и frontend-разработчиков.

Thymeleaf был разработан с учетом стандартов Web, особенно HTML5, что позволяет создавать полностью проверенные шаблоны. Обособленно, без запуска приложения, шаблон можно проверить HTML-валидатором на соответствие стандарту HTML5. Шаблон может быть открыт в браузере как обычный html-файл (чем он и является), при этом он будет правильно отображен как валидная веб-страница. Thymeleaf обеспечивает интеграцию со Spring Framework.

### Типы используемых шаблонов

Из коробки Thymeleaf позволяет обрабатывать шесть режимов шаблонов:

* HTML;
* XML;
* TEXT;
* JAVASCRIPT;
* CSS;
* RAW.

По умолчанию используется режим HTML.

### Диалекты Thymeleaf

Чтобы добиться более простой и удобной интеграции, Thymeleaf предоставляет диалект Thymeleaf Spring, который специально реализует все необходимые функции для правильной работы со Spring.

Официальные пакеты интеграции **thymeleaf-spring3** и **thymeleaf-spring4** определяют диалект **SpringStandard Dialect**, который в основном совпадает со Standard Dialect, но содержит и небольшие изменения, позволяющие лучше использовать некоторые функции Spring Framework.

Помимо всех функций, уже присутствующих в стандартном диалекте и, следовательно, унаследованных, в диалекте **SpringStandard Dialect** представлены следующие особенности:

* в качестве языка выражений используется **Spring Expression Language** (SpEL), а не OGNL. Поэтому все выражения вида **${...}** и **\*{...}** будут вычислены с помощью SpEL;
* есть доступ к любому бину в приложении с использованием SpEL, например **${@myBean.doSomething()}**;
* помимо новой реализации **th:object** имеются новые атрибуты для обработки формы: **th:field**, **th:errors** и **th:errorclass**.

Мы будем рассматривать диалект Thymeleaf Spring, так как он позволяет использовать Thymeleaf как полную замену JSP в приложениях Spring.

Thymeleaf — это механизм шаблонов Java для обработки и создания HTML, XML, JavaScript, CSS и текста. На этом уроке мы обсудим, как работать с Thymeleaf и Spring, рассмотрим базовые варианты использования на уровне представления приложения Spring MVC. Библиотека Thymeleaf расширяема, и ее естественная возможность шаблонирования гарантирует, что шаблоны могут быть прототипированы независимо — без использования сервера приложений. Это делает разработку очень быстрой по сравнению с другими популярными движками шаблонов, такими как JSP.

Шаблоны Thymeleaf выглядят как валидный статический HTML. В работающем приложении атрибуты пространства имен **th:** будут динамически вычислены и представлены как тело тега. Или будет выполнено дополнительное действие, например, как цикл ниже:

|  |
| --- |
| <html lang="en" xmlns:th="<http://www.thymeleaf.org>">  ...  <table>    <thead>      <tr>        <th th:text="#{msgs.headers.name}">Name</th>        <th th:text="#{msgs.headers.price}">Price</th>      </tr>    </thead>    <tbody>      <tr th:each="prod: ${allProducts}">        <td th:text="${prod.name}">Oranges</td>        <td th:text="${#numbers.formatDecimal(prod.price, 1, 2)}">0.99</td>      </tr>    </tbody>  </table> |

### Интеграция со Spring

Thymeleaf предлагает набор возможностей для интеграций со Spring, чтобы использовать его как полнофункциональную замену JSP в приложениях Spring MVC. Это позволяет:

* создавать сопоставленные методы в объектах Spring MVC — **@Controller** для шаблонов, управляемых Thymeleaf;
* использовать Spring Expression Language (Spring EL) в шаблонах;
* создавать формы в шаблонах, которые полностью интегрированы с бинами обработки форм;
* реализовывать интернационализацию сообщений с помощью их файлов, управляемых Spring;
* маршрутизировать шаблоны, используя собственные механизмы разрешения ресурсов Spring.

Чтобы включить Thymeleaf в проект, требуется библиотека **thymeleaf-spring**. Для этого необходимо добавить следующие зависимости к файлу **Maven pom.xml**:

|  |
| --- |
| <dependency>     <groupId>org.thymeleaf</groupId>     <artifactId>thymeleaf-spring5</artifactId>     <version>3.0.10.RELEASE</version>  </dependency> |

Класс **SpringTemplateEngine** выполняет все этапы настройки. Вы можете настроить этот класс как компонент в конфигурационном файле Java:

|  |
| --- |
| @Bean   public SpringResourceTemplateResolver templateResolver() {     SpringResourceTemplateResolver templateResolver = new SpringResourceTemplateResolver ();     viewResolver.setPrefix("/WEB-INF/templates/");     viewResolver.setSuffix(".html");     return templateResolver;   }    @Bean   public TemplateEngine templateEngine() {     TemplateEngine templateEngine = new SpringTemplateEngine();     templateEngine.setTemplateResolver(templateResolver());     return templateEngine;   }    @Bean   public ThymeleafViewResolver thymeleafViewResolver() {     ThymeleafViewResolver thymeleafViewResolver = new ThymeleafViewResolver ();     thymeleafViewResolver.setTemplateEngine(templateEngine());     thymeleafViewResolver.setEncoding("UTF-8");     return thymeleafViewResolver;   } |

Свойства **prefix** и **suffix** бина **templateResolver** указывают расположение файлов шаблонов внутри (относительно) папки **webapp** и расширение файлов. Интерфейс **ViewResolver** в Spring MVC транслирует имена представлений, возвращаемые контроллером, в реальные объекты просмотра. **ThymeleafViewResolver** реализует интерфейс **ViewResolver** и используется, чтобы определять виды Thymeleaf для рендеринга с учетом имени представления.

### Вариация Spring Boot

В случае Spring Boot проекта Thymeleaf добавляется соответствующим стартером:

|  |
| --- |
| <dependency>    <groupId>org.springframework.boot</groupId>    <artifactId>spring-boot-starter-thymeleaf</artifactId>  </dependency> |

Кроме библиотеки стартер добавляет небольшой «мешочек магии» в виде класса [ThymeleafAutoConfiguration](https://github.com/spring-projects/spring-boot/blob/v1.5.4.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/thymeleaf/ThymeleafAutoConfiguration.java), что избавляет нас написания вручную бинов конфигурации, рассмотренных выше. А класс [ThymeleafProperties](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/thymeleaf/ThymeleafProperties.java) определяет по умолчанию следующие опции, конфигурируемые в файле **application.properties**:

|  |
| --- |
| # THYMELEAF (ThymeleafAutoConfiguration)  spring.thymeleaf.cache=true                   # кеширование  # Check template exists before rendering it.  spring.thymeleaf.check-template=true  # Check templates location exists.  spring.thymeleaf.check-template-location=true  spring.thymeleaf.content-type=text/html       # Content-Type value.  # Enable MVC Thymeleaf view resolution.  spring.thymeleaf.enabled=true  spring.thymeleaf.encoding=UTF-8               # Template encoding.  # список отключенных представлений  spring.thymeleaf.excluded-view-names=  spring.thymeleaf.mode=HTML5                   # режим шаблонов  spring.thymeleaf.prefix=classpath:/templates/ # путь к шаблонам, Prefix  # расширение файлов шаблонов, Suffix  spring.thymeleaf.suffix=.html  spring.thymeleaf.template-resolver-order=     # порядок поиска в цепочке  # закрытый список используемых шаблонов  spring.thymeleaf.view-names= |

### Отображение строк из файлов message.properties и интернационализация

Атрибут **th:text="# {key}"** может использоваться для отображения значений из файлов свойств. Для этого файл свойств должен быть указан при конфигурировании бина **messageSource**:

|  |
| --- |
| @Bean  @Description("Spring Message Resolver")  public ResourceBundleMessageSource messageSource() {    ResourceBundleMessageSource messageSource = new                                               ResourceBundleMessageSource();    messageSource.setBasename("messages");    return messageSource;  } |

Отметим, что Spring Boot выполняет данное конфигурирование самостоятельно.

Код HTML-шаблона для отображения значения, связанного с ключом **welcome.message**:

|  |
| --- |
| <span th:text="#{welcome.message}" /> |

Строки сообщений могут быть параметризированы:

|  |
| --- |
| <span th:text="#{welcome.message(user.name)}" /> |

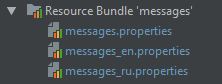
Файл **messages.properties**:

|  |
| --- |
| welcome.message=Welcome {0}! |

По умолчанию приложение **Spring Boot** будет искать файлы сообщений, содержащие ключи, и значения интернационализации в папке **src/main/resources**.

Файл для локали по умолчанию будет иметь имя **messages.properties**, а файлы для каждой локали — **messages\_XX.properties**, где **XX** — код локали. Если файла с запрошенным кодом не существует, будет использован файл локали по умолчанию.

Интернационализация достигается добавлением суффикса языкового кода к имени файла свойств:



Файл **messages\_ru.properties**:

|  |
| --- |
| welcome.message=Добро пожаловать {0}! |

### Отображение атрибутов модели

Атрибут **th:text = "${attributename}"** используется для отображения значения атрибутов модели. Добавим атрибут **model** с именем **serverTime** в класс контроллера:

|  |
| --- |
| model.addAttribute("serverTime", dateFormat.format(new Date())); |

Код HTML-шаблона для отображения значения атрибута **serverTime**:

|  |
| --- |
| Current time is <span th:text="${serverTime}" /> |

### Отображение атрибутов коллекций

Если атрибут **model** представляет собой коллекцию объектов, атрибут **th:each** может использоваться для перебора этой коллекции. Определим класс модели **Item**:

|  |
| --- |
| // новость: шапка. текст, дата, источник  public class Item {    private final String header;    private final String text; |

Теперь определим список новостей как атрибут модели в классе контроллера:

|  |
| --- |
| // ... логика наполнения списка опущена  @ModelAttribute("items")  public List<Item> populateItems() {    return items;  } |

Используем шаблон **Thymeleaf**, чтобы вывести список элементов и отобразить значения полей каждого пользователя:

|  |
| --- |
| <table>    <tr th:each="item: ${items}">      <td th:text="${item.header}"/>      <td th:text="${item.text}"/>    </tr>  </table> |

### Форматированный вывод

Для форматированного вывода пользовательских типов можно воспользоваться конструкцией **{{...}}**:

|  |
| --- |
| <td th:text="$**{{**item.date**}}**"/> |

И определить бин пользовательского форматтера в конфигурации приложения:

|  |
| --- |
| @SpringBootApplication  public class ThymeleafApplication {    ...    @Bean    public DateFormatter dateFormatter() {      return new DateFormatter();    } |

|  |
| --- |
| public class DateFormatter implements Formatter<Date> {    @Autowired    private MessageSource messageSource;    public Date parse(final String text, final Locale locale)                        throws ParseException {      final SimpleDateFormat dateFormat = createDateFormat(locale);      return dateFormat.parse(text);    }    public String print(final Date object, final Locale locale) {      final SimpleDateFormat dateFormat = createDateFormat(locale);      return dateFormat.format(object);    }    private SimpleDateFormat createDateFormat(final Locale locale) {      final String format = this.messageSource.getMessage("date.format",                                                          null, locale);      final SimpleDateFormat dateFormat = new SimpleDateFormat(format);      dateFormat.setLenient(false);      return dateFormat;    }  } |

### Обработка форм

Действие формы может быть указано с помощью атрибута **th:action**. Атрибут **th:object** связывает данную форму с указанным объектом модели. Отдельные поля отображаются с использованием атрибута **th:field="\*{name}"**, где **name** — имя свойства объекта, для получения и заполнения которого будет использованы геттеры и сеттеры.

Следующий пример шаблона:

|  |
| --- |
| <form action="#" th:action="@{/some}" th:object="${item}" method="post">    <input type="text" th:field="\*{date}" />    <input type="text" th:field="\*{header}" />    <input type="text" th:field="\*{text}" />  </form> |

… отображается в:

|  |
| --- |
| <form action="/some" method="post">    <input type="text" id="date" name="date" value="10.10.2017" />    <input type="text" id="header" name="header" value="" />    <input type="text" id="text" name="text" value="" />  </form> |

Заметим на примере свойства **date**, что **th:field** автоматически использует доступные форматтеры пользовательских типов.

### Валидация полей формы

Функция **#fields.hasErrors()** может использоваться для проверки наличия ошибок. Функция **#fields.errors()** отображает ошибки определенного поля. Для обеих этих функций входным параметром будет имя поля — в его качестве могут выступать предопределенные константы «**\***» и «**all**».

Пример:

|  |
| --- |
| <input type="text" th:field="\*{date}" />  <p th:if="${#fields.hasErrors('date')}" th:errors="\*{date}">Incorrect date</p> |

|  |
| --- |
| <ul>      <li th:each="err : ${#fields.errors('all')}" th:text="${err}" />  </ul> |

### Упрощение стилизации CSS на основе ошибок

Применительно к тегу поля формы (**input**, **select**, **textarea**...) атрибут **th:errorclass** будет читать имя поля, которое должно быть проверено, из любого существующего имени или **th:** атрибутов поля в том же теге. А затем добавит указанный класс CSS в тег, если в этом поле есть связанные ошибки.

Пример:

|  |
| --- |
| <input type="text" th:field="\*{date}" th:errorclass="error" /> |

Если в поле будет ошибка, **date** будет отображено, как предписывает следующий HTML-код:

|  |
| --- |
| <input type="text" id="date" name="date" value="2017-10-10" class="error" /> |

### Условные выражения

Атрибут **th:if=”${condition}”** используется для отображения раздела представления, если условие истинно.

Пример:

|  |
| --- |
| <div id="comments" th:if="${!items.empty}">      <h3>Comments</h3>      <ul th:each="item : ${items}">          <li><span th:text="${item}">comment</span></li>      </ul>  </div> |

**<div>** и все его содержимое будет отображено только при непустом списке **items**.

Атрибут **th:unless=”${condition}”** — это антипод **th:if**. Действие будет выполнено при ложном значении условия.

Пример:

|  |
| --- |
| <a href="comments.html" th:href="@{/comments}" th:unless="${items.empty}">view comments</a> |

Ссылка **<a href=...>** будет отображаться только до тех пор (**unless**), пока **items** не пуст.

# Практическое задание

1. Добавить навигацию и пагинацию по страницам в таблице товаров.
2. Реализовать фильтр товаров по диапазону цен на основе двух полей (минимальная и максимальная цена) и кнопки («Фильтровать»).
3. \* Добавить возможность редактировать существующие товары.

# Дополнительные материалы

1. <https://www.thymeleaf.org/documentation.html>

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Официальная документация. Tutorial: Using Thymeleaf](http://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html).
2. [Официальная документация. Tutorial: Thymeleaf + Spring](http://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html).

**Spring Boot**

Spring Boot. Архитектура. Модель. Обзор решений.

**Оглавление**

[Spring Boot](https://docs.google.com/document/d/1Nn6jHhibY4Zwmuc1Acp_mciWrvvGqkgdT8DIWoXEYtk/edit#heading=h.x48254cd671k)

[Быстрая разработка приложений со Spring Boot](https://docs.google.com/document/d/1Nn6jHhibY4Zwmuc1Acp_mciWrvvGqkgdT8DIWoXEYtk/edit#heading=h.2s8eyo1)

[Стартеры](https://docs.google.com/document/d/1Nn6jHhibY4Zwmuc1Acp_mciWrvvGqkgdT8DIWoXEYtk/edit#heading=h.17dp8vu)

[Указание версии Java](https://docs.google.com/document/d/1Nn6jHhibY4Zwmuc1Acp_mciWrvvGqkgdT8DIWoXEYtk/edit#heading=h.3rdcrjn)

[Spring Boot Maven plugin](https://docs.google.com/document/d/1Nn6jHhibY4Zwmuc1Acp_mciWrvvGqkgdT8DIWoXEYtk/edit#heading=h.26in1rg)

[Свойства (application.properties)](https://docs.google.com/document/d/1Nn6jHhibY4Zwmuc1Acp_mciWrvvGqkgdT8DIWoXEYtk/edit#heading=h.35nkun2)

[Особенности различных применений](https://docs.google.com/document/d/1Nn6jHhibY4Zwmuc1Acp_mciWrvvGqkgdT8DIWoXEYtk/edit#heading=h.1ksv4uv)

[Статический контент](https://docs.google.com/document/d/1Nn6jHhibY4Zwmuc1Acp_mciWrvvGqkgdT8DIWoXEYtk/edit#heading=h.44sinio)

[Практическое задание](https://docs.google.com/document/d/1Nn6jHhibY4Zwmuc1Acp_mciWrvvGqkgdT8DIWoXEYtk/edit#heading=h.pojp55rqdikq)

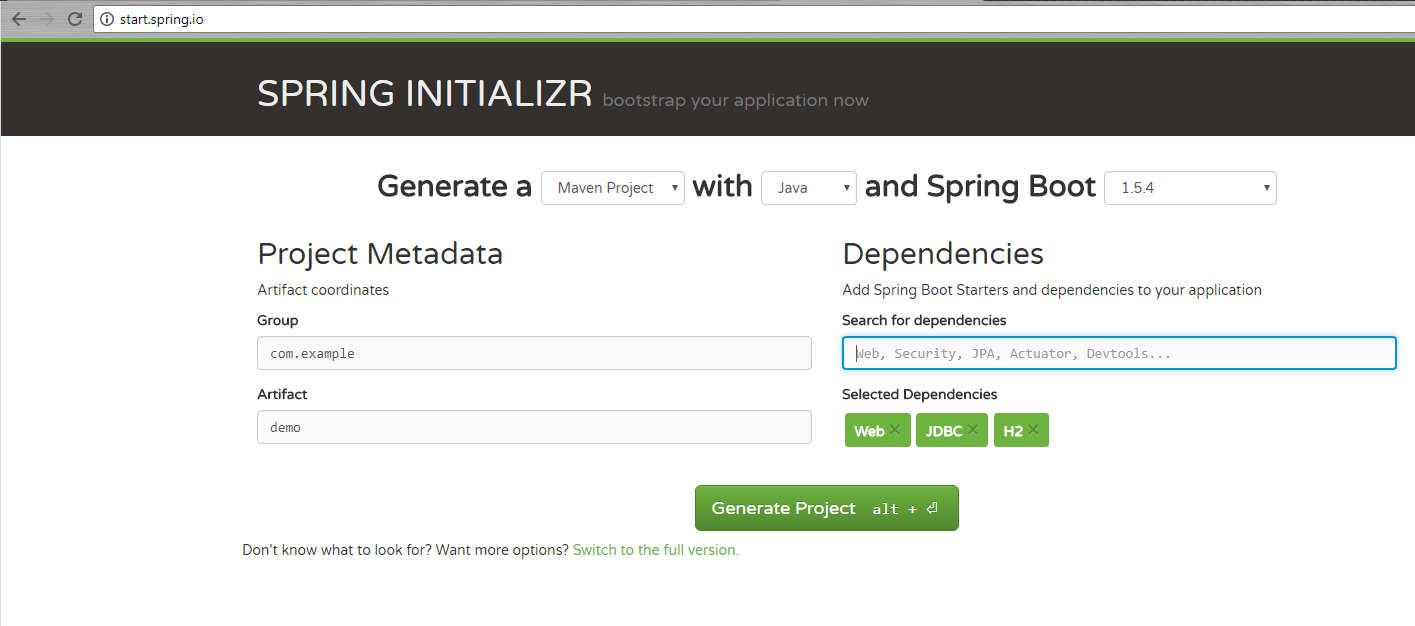
[Дополнительные материалы](https://docs.google.com/document/d/1Nn6jHhibY4Zwmuc1Acp_mciWrvvGqkgdT8DIWoXEYtk/edit#heading=h.3whwml4)

[Используемая литература](https://docs.google.com/document/d/1Nn6jHhibY4Zwmuc1Acp_mciWrvvGqkgdT8DIWoXEYtk/edit#heading=h.2bn6wsx)

# Spring Boot

Spring Boot упрощает создание Spring-приложений: для большинства из них требуется очень небольшая настройка конфигурации, но при этом их можно настраивать по своему усмотрению. С помощью Spring Boot можно создавать Java-приложения, которые запускаются из командной строки через java — jar, или более традиционно — путем развертывания war-архива на сервере приложений.

Для того, чтобы создать с нуля полноценное Spring Boot приложение можно воспользоваться сервисом start.spring.io — это веб-сервис, призванный построить каркас приложения с определенной структурой каталогов и файлов, которая понятна современным средствам автоматизированной сборки проектов (например, Gradle или Maven).



После генерации проекта сервисом получаем zip-архив, содержащий его готовую файловую структуру:

1. Файл **pom.xml** — файл описания проекта.
2. Папка **src** — содержит все исходные файлы.
3. Папка **src/main** — исходные файлы разрабатываемого приложения.
4. Папка **src/main/java** — исходный Java-код.
5. Папка **src/main/resources** — файлы, использующиеся при компиляции и исполнении.
6. Папка **src/test** — исходные файлы для автоматического тестирования.
7. Папка **src/test/java** — исходные файлы Java для автоматического тестирования.

В pom-файле, описывающем проект, видим и стартер по умолчанию, и запрошенные ранее зависимости.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.example</groupId>  <artifactId>demo</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>  <name>demo</name>  <description>Demo project for Spring Boot</description>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.1.3.RELEASE</version>  <relativePath/>  </parent>  <properties>    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  <dependency>  <groupId>com.h2database</groupId>  <artifactId>h2</artifactId>  <scope>runtime</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

Инициализатор **Spring Boot** внутри **Intellij IDEA Ultimate** также использует описанный выше сервис, чтобы создать файл **pom.xml** для нового Spring-проекта.

## Быстрая разработка приложений со Spring Boot

Из стартового набора рассмотрим еще два файла:

* **DemoApplication.java** — класс начальной загрузки и конфигурации Spring;
* **application.properties** — файл настройки свойств Spring Boot.

|  |
| --- |
| @RestController  @SpringBootApplication  public class DemoApplication {    public static void main(String[] args) {      SpringApplication.run(DemoApplication.class, args);    }    @RequestMapping("/\*\*")    public String helloWorld() {      return "Hello World";    }  } |

У класса **DemoApplication** двойное назначение:

1. Получает управление при запуске **jar**.
2. Отвечает за конфигурирование приложения благодаря аннотации **@SpringBootApplication**.

Аннотация **@SpringBootApplication** в числе прочих наследуется от аннотаций **@Configuration**, **@ComponentScan**, **@EnableAutoConfiguration**:

* **@Configuration** — объявляет класс классом Java-based конфигурации;
* **@ComponentScan** — позволяет выполнять компонентное сканирование;
* **@EnableAutoConfiguration** — это вызов «магии»: она сообщает Spring Boot, что необходимо «угадать», как вы хотите настроить фреймворк, основываясь на добавленных зависимостях.

Поскольку стартер **spring-boot-starter-web** добавил **Tomcat** и **Spring MVC**, автоконфигурация предполагает, что вы разрабатываете веб-приложение.

### Стартеры

Стартеры — это наборы удобных дескрипторов зависимостей, которые можно включить в приложение. Например, чтобы использовать Spring и JPA для доступа к базе данных, нужно просто добавить зависимость **spring-boot-starter-data-jpa** в проект. Стартеры содержат множество зависимостей, необходимых для быстрого запуска проекта с помощью согласованного и поддерживаемого набора управляемых транзитивных зависимостей. Все официальные стартеры следуют единой схеме именования: **spring-boot-starter-\***, где **\*** — конкретный тип приложения.

 Вот несколько популярных стартеров Spring Boot:

* **spring-boot-starter-web** — используется для создания веб-служб RESTful с использованием Spring MVC и Tomcat в качестве встроенного контейнера приложений;
* **spring-boot-starter-thymeleaf** — подключение шаблонизатора Thymeleaf;
* **spring-boot-starter-data-jpa** — подключает модуль Spring Data JPA;
* **spring-boot-starter-security** — подключает модуль Spring Security для обеспечения безопасности веб-приложения;
* **spring-boot-starter-jersey** — альтернатива Spring-boot-starter-web, в которой используется встроенный сервер приложений Jersey, а не Tomcat;
* **spring-boot-starter-jdbc** — реализует пул соединений JDBC, основан на реализации пула JDBC Tomcat.

Вернемся к pom-файлу, в котором мы наследуем свой проект от специального стартера:

|  |
| --- |
| ...  <parent>    <groupId>org.springframework.boot</groupId>    <artifactId>spring-boot-starter-parent</artifactId>    <version>2.1.3.RELEASE</version>  </parent>  ... |

**spring-boot-starter-parent** — это специальный стартер, предоставляющий настройки Maven по умолчанию и раздел управления зависимостями, чтобы вы могли опустить теги версии для Spring-зависимостей.

### Указание версии Java

По-умолчанию, в **spring-boot-starter-parent** указана совместимость с Java 6. Для того, чтобы изменить версию на более новую, достаточно в pom.xml в элементе <properties> прописать элемент <java.properties> с соответствующей версией.

|  |
| --- |
| <properties>    <java.version>1.8</java.version>  </properties> |

### Spring Boot Maven plugin

По умолчанию инициализатор **Spring Boot** в секции **<plugins>** pom-файла указывает плагин **spring-boot-maven-plugin**:

|  |
| --- |
| <build>    <plugins>      <plugin>        <groupId>org.springframework.boot</groupId>        <artifactId>spring-boot-maven-plugin</artifactId>      </plugin>    </plugins>  </build> |

Данный плагин решает несколько полезных задач:

* перепаковывает выходной jar-файл проекта в исполняемый **uber-jar**, включающий в себя все jar-файлы зависимостей проекта, что очень удобно при развертывании;
* находит точку входа в проекте — класс, содержащий функцию **main()**, указывает его в манифесте переупакованного jar-файла (uber-jar);

|  |
| --- |
| Main-Class: org.springframework.boot.loader.JarLauncher  Start-Class: com.example.demo.DemoApplication |

* при указании war-упаковки перепаковывает выходной war-файл проекта в war-файл, включающий в себя все jar-файлы зависимостей;

|  |
| --- |
| <packaging>war</packaging> |

* предоставляет определитель версии зависимостей, который устанавливает номер версии для соответствия зависимостям Spring Boot. Это избавляет от необходимости указывать их версию. Кроме того, вы всегда можете переопределить данное поведение и указать версию по своему усмотрению.

### Свойства (application.properties)

Наследование от **spring-boot-starter-parent** включает использование файла свойств **application.properties**, в котором могут быть указаны свойства проекта — например, порт, на котором слушает встроенный web-сервер, или настройки Spring-бинов.

Указание 8189-го порта (а не на 8080, как указано по умолчанию) выглядит следующим образом:

|  |
| --- |
| server.port=8189 |

Ранее для гибкого указания свойств JDBC-соединения (не посредством Java-конфигурирования) приходилось описывать бины в файле **applicationContext.xml**. Так по-прежнему можно делать, если вы привыкли или любите XML.

Spring Boot, зная, что вы используете реализацию JDBC-драйвера (в нашем примере это **H2 Database : com.h2database**), сам создает бины DataSource, свойства которых выставляет по умолчанию. Чтобы выставить другие значения, их можно прописать в файле свойств.

Например, по умолчанию H2-драйвер использует URL-подключения к in-memory БД **jdbc:h2:mem:testdb**. Указываем в файле свойств:

|  |
| --- |
| spring.datasource.url=jdbc:h2:~/spring.h2  spring.datasource.driver-class-name=org.h2.Driver |

# Особенности различных применений

## Статический контент

По умолчанию Spring Boot будет отдавать статический контент из каталога **/static** (или **/public**, или **/resources**, или **/META-INF/resources**) в **classpath** или из каталога **ServletContext**. Он использует **ResourceHttpRequestHandler** из Spring MVC, поэтому вы можете изменить это поведение, добавив свой **WebMvcConfigurerAdapter** и переопределив метод **addResourceHandlers**.

По умолчанию статический контент отображается на корень (/\*\*) веб-приложения с точки зрения браузера. Но можно настроить иное отображение, используя свойство **spring.mvc.static-path-pattern**. Например, для отображения статического контента в **/resources/\*\***:

|  |
| --- |
| spring.mvc.static-path-pattern=/resources/\*\* |

Физическое расположение статического контента можно указать, используя свойство **spring.resources.static-locations**. Можно указать список из нескольких локаций.

В дополнение к обычным статическим ресурсам можно использовать упакованные в формате [WebJar](https://www.webjars.org/documentation#springboot) Для этого необходимо дополнить pom.xml соответствующими зависимостями:

|  |
| --- |
| <dependencies>    ...    <dependency>      <groupId>org.webjars</groupId>      <artifactId>jquery</artifactId>      <version>2.1.1</version>    </dependency>    <dependency>      <groupId>org.webjars</groupId>      <artifactId>bootstrap</artifactId>      <version>3.2.0</version>    </dependency>    ...  </dependencies> |

Далее мы можем ссылаться на эти статические ресурсы следующим образом:

|  |
| --- |
| <link rel='stylesheet' href='/webjars/bootstrap/3.2.0/css/bootstrap.min.css'>  <script type="text/javascript" src="/webjars/bootstrap/3.2.0/js/bootstrap.min.js"></script> |

Заметьте, что версию внешнего ресурса (js-библиотеки, css-файлов) приходится указывать ссылкой на ресурс и следить, чтобы она соответствовала версии, указанной в зависимости, — а это неудобно. Решение — подключить в зависимость локатор JS/CSS-библиотек:

|  |
| --- |
| <dependency>    <groupId>org.webjars</groupId>    <artifactId>webjars-locator</artifactId>    <version>RELEASE</version>  </dependency> |

Теперь указание версии можно опустить:

|  |
| --- |
| <link rel='stylesheet' href='/webjars/bootstrap/css/bootstrap.min.css'>  <script type="text/javascript" src="/webjars/bootstrap/js/bootstrap.min.js"></script> |

# Практическое задание

1. Перенести функциональность, реализованную на прошлых занятиях, на платформу Spring Boot.

# Дополнительные материалы

1. [Обратная сторона Spring](https://habrahabr.ru/post/334448/).
2. [Spring Boot. Boot up your development (Sergey Morenets)](https://youtu.be/uRXRIDy-6UU).

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Официальная документация проекта Spring Boot](http://docs.spring.io/spring-boot/docs/1.5.4.RELEASE/reference/htmlsingle).
2. [Spring Boot basics](https://www.ibm.com/developerworks/library/j-spring-boot-basics-perry/index.html).

**Spring REST**

Spring REST. HTTP 1.1. CRUD-операции.

**Оглавление**

[Введение](https://docs.google.com/document/d/1WnXJizgO92LJAhrxC29OoXOhUUeZGX5jo9XASxuR1cg/edit#heading=h.3znysh7)

[Формат JSON](https://docs.google.com/document/d/1WnXJizgO92LJAhrxC29OoXOhUUeZGX5jo9XASxuR1cg/edit#heading=h.lo1ruj9w45h4)

[Архитектура REST](https://docs.google.com/document/d/1WnXJizgO92LJAhrxC29OoXOhUUeZGX5jo9XASxuR1cg/edit#heading=h.c9jkinikee7t)

[Подготовка к работе](https://docs.google.com/document/d/1WnXJizgO92LJAhrxC29OoXOhUUeZGX5jo9XASxuR1cg/edit#heading=h.7xf48hytxi2d)

[CRUD в REST](https://docs.google.com/document/d/1WnXJizgO92LJAhrxC29OoXOhUUeZGX5jo9XASxuR1cg/edit#heading=h.x5is4npunu1e)

[Реализация контроллера](https://docs.google.com/document/d/1WnXJizgO92LJAhrxC29OoXOhUUeZGX5jo9XASxuR1cg/edit#heading=h.2tn5doquomlz)

[MapStruct](https://docs.google.com/document/d/1WnXJizgO92LJAhrxC29OoXOhUUeZGX5jo9XASxuR1cg/edit#heading=h.31o84z307l8)

[Практическое задание](https://docs.google.com/document/d/1WnXJizgO92LJAhrxC29OoXOhUUeZGX5jo9XASxuR1cg/edit#heading=h.1t3h5sf)

[Дополнительные материалы](https://docs.google.com/document/d/1WnXJizgO92LJAhrxC29OoXOhUUeZGX5jo9XASxuR1cg/edit#heading=h.4d34og8)

[Используемая литература](https://docs.google.com/document/d/1WnXJizgO92LJAhrxC29OoXOhUUeZGX5jo9XASxuR1cg/edit#heading=h.2s8eyo1)

**Введение**

В REST-сервисах запросы и ответы строятся вокруг передачи представлений ресурсов, которые определяются глобальными идентификаторами — обычно унифицированными идентификаторами ресурсов (Uniform Resource Identifier — URI). Клиентские же приложения используют для управления ресурсами HTTP-методы (такие как GET, POST, PUT, DELETE).

Например, ответом на GET-запрос <http://localhost:8080/app/students/15> будет представление студента с ID = 15, которое содержит информацию о студенте в формате JSON/XML. Вид представления зависит от реализации на стороне сервера и MIME-типа запроса клиента.

Веб-сервисы RESTful реализуются с использованием HTTP и принципов REST. До появления поддержки в Spring для создания REST-сервисов в Java использовались библиотеки Restlet, RestEasy и Jersey.

**Формат JSON**

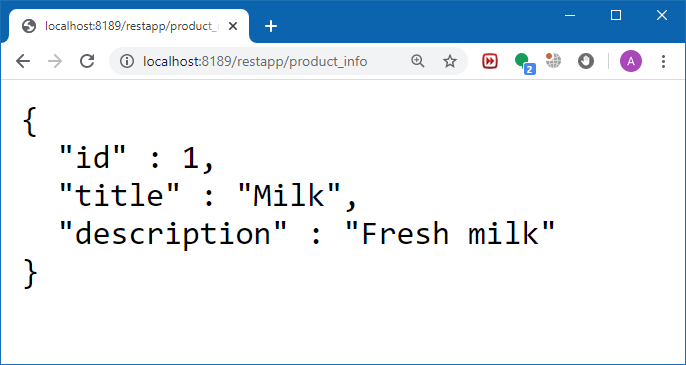
Давайте посмотрим как в обычном Spring Boot приложении работать с JSON представлением объектов. Допустим у нас есть простой класс Product, который не является сущностью, то есть нам для работы с продуктом не придется обращаться к базе данных.

|  |
| --- |
| **public class** Product {  **private** Long **id**;  **private** String **title**;  **private** String **description**;     // Геттеры и сеттеры  **public** Product(Long id, String title, String description) {  **this**.**id** = id;  **this**.**title** = title;  **this**.**description** = description;     }  } |

Мы хотим чтобы в ответ на GET запрос, веб-сервис прислал бы объект типа Product в формате JSON. Для этого создадим ProductController и пропишем в нем следующий код.

|  |
| --- |
| @Controller  **public class** ProductController {     @GetMapping(**"/product\_info"**)     @ResponseBody  **public** Product getOneProduct() {  **return new** Product(1L, **"Milk"**, **"Fresh milk"**);     }  } |

Если раньше в ответ на наш запрос контроллер возвращал имя представления (view), и потом пользователю приходила html-страница, собранная по этому представлению. То в данном случае аннотация @ResponseBody говорит о том, что в ответ на запрос, сервер отправит клиенту указанный объект, и зашьет его в тело ответа. Вот так результат будет выглядеть в браузере.



В чем преимущество формата JSON - это текстовый формат представления объектов, который поддерживается многими языками программирования “из коробки”. А если вдруг подходящей библиотеки найти не удалось, то можно написать и свой парсер. Поскольку структура и состояние объектов описывается в текстовом виде, вы можете передавать объекты между сервисами, написанными на разных языках программирования.

Каким образом Spring преобразовал наш объект в JSON? В Spring Boot интегрирована библиотека Jackson, которая берет на себя задачу сериализации и десериализации в/из формата JSON.

Давайте разберемся в том, по каким правилам объекты преобразуются в JSON. Если у нас есть класс Product, имеющий числовое поле id, и два строковых поля title и description, то получим:

|  |
| --- |
| {    "id": 1,    "title": "Milk",    "description": "Fresh milk"  } |

Фигурные скобки означают сам объект, класс при этом не указывается. Внутри скобок прописываются пары ключ-значение, обозначающие поле объекта и его значение, каждая такая пара разделяется двоеточием с пробелом. Имя ключа заключено в двойные кавычки. Если значение числовое, то записывается без кавычек, если значение текстовое, то оно берётся в двойные кавычки. Если в объекте есть числовой массив, то его содержимое берется в квадратные скобки.

|  |
| --- |
| {    "values": [ 1, 2, 3, 4, 5 ]  } |

Если один объект ссылается на другой объект, например, экземпляр класса Пользователь содержит поле типа Адрес, то получим запись вида:

|  |
| --- |
| {    "firstName": "Alexander",    "age": 30,    "address": {      "city": Rostov-on-Don,      "street": Central st. 1    }  } |

Любые объекты строятся по правилам, описанным выше, как видите формат достаточно простой. Разобравшись с тем, как можно передавать любые объекты в формате JSON,

**Архитектура REST**

**REST** (**Re**presentational **S**tate **T**ransfer - “передача состояния представления”) - архитектурный стиль построения распределенных веб-приложений, предложенный Роем Филдингом в его диссертации “*Architectural Styles and the Design of Network-based Software Architectures*”.

**REST** использует клиент-серверную архитектуру. Клиент и сервер решают разные задачи. Сервер хранит и/или манипулирует информацией и делает ее доступной для клиента. Клиент берет эту информацию и отображает ее пользователю и/или использует ее для выполнения последующих запросов. Такое разделение задач позволяет как клиенту, так и серверу развиваться независимо, так как требуется только то, чтобы интерфейс оставался прежним.

**REST** не хранит состояние (**stateless**). Это означает, что “общение” между клиентом и сервером всегда содержит всю информацию, необходимую для выполнения запросов. На сервере не сессия работы с клиентом, она полностью хранится на стороне клиента. Если для доступа к ресурсу требуется аутентификация, клиент должен аутентифицировать себя при каждом запросе.

**REST** кешируется. Клиент, сервер и любые промежуточные компоненты могут кэшировать ресурсы для повышения производительности.

**REST** обеспечивает единый интерфейс между компонентами. Это упрощает архитектуру, так как все компоненты используют одни и те же правила для общения друг с другом. Это также облегчает понимание взаимодействия между различными компонентами системы. Для этого требуется ряд ограничений. Они рассматриваются в остальной части главы.

**REST** имеет многослойную структуру. Отдельные компоненты не могут видеть за пределами непосредственного слоя, с которым они взаимодействуют. Это означает, что клиент, подключающийся к промежуточному компоненту, например прокси, не знает, что находится за его пределами. Это позволяет компонентам быть независимыми и, следовательно, легко заменяемыми или расширяемыми.

**Подготовка к работе**

Для разбора принципов работы с REST-сервисами на Spring необходимо подготовить базовый проект: подключить зависимости (**spring-boot-web-starter**, **spring-boot-starter-data-jpa**, **postgresql**). Проект работает со списком студентов и книг, для которых необходимо подготовить классы-сущности и скрипты для генерации таблиц в базе данных.

|  |
| --- |
| <**dependency**>    <**groupId**>org.springframework.boot</**groupId**>    <**artifactId**>spring-boot-starter-data-jpa</**artifactId**> </**dependency**> <**dependency**>    <**groupId**>org.springframework.boot</**groupId**>    <**artifactId**>spring-boot-starter-web</**artifactId**> </**dependency**> <**dependency**>    <**groupId**>org.postgresql</**groupId**>    <**artifactId**>postgresql</**artifactId**>    <**scope**>runtime</**scope**>  </**dependency**> |

SQL-скрипты для тестовой базы данных:

|  |
| --- |
| **DROP TABLE** IF **EXISTS** students;  **CREATE TABLE** students (     id bigserial **NOT NULL**,     name **varchar**(100) **NOT NULL**,  **PRIMARY KEY**(id)  );  **INSERT INTO** students (name) **VALUES** (**'Bob'**), (**'John'**), (**'Michael'**);  **DROP TABLE** IF **EXISTS** books;  **CREATE TABLE** books (     id bigserial **NOT NULL**,     title **varchar**(100) **NOT NULL**,  **PRIMARY KEY**(id)  );  **DROP TABLE** IF **EXISTS** students\_books;  **CREATE TABLE** students\_books (    student\_id bigint **NOT NULL**,     book\_id bigint **NOT NULL**,  **PRIMARY KEY** (student\_id, book\_id),  **FOREIGN KEY** (student\_id) **REFERENCES** students (id),  **FOREIGN KEY** (book\_id) **REFERENCES** books (id)  );  **INSERT INTO** books (title) **VALUES** (**'Harry Potter'**), (**'Lord Of The Ring'**); |

Классы сущности:

|  |
| --- |
| @Entity @Table(name = **"students"**) **public** **class** Student {   @Id   @Column(name = **"id"**)   @GeneratedValue(strategy = GenerationType.IDENTITY)   **private** Long id;    @Column(name = **"name"**)   **private** String name;    @ManyToMany   @JoinTable(     name = **"students\_books"**,     joinColumns = @JoinColumn(name = **"student\_id"**),     inverseJoinColumns = @JoinColumn(name = **"book\_id"**)   )   **private** List<Book> books;    *// Геттеры и сеттеры*    **public** Student() {   } }  @Entity @Table(name = **"books"**) **public** **class** Book {   @Id   @Column(name = **"id"**)   @GeneratedValue(strategy = GenerationType.IDENTITY)   **private** Long id;    @Column(name = **"title"**)   **private** String title;    @ManyToMany   @JoinTable(     name = **"students\_books"**,     joinColumns=@JoinColumn(name = **"book\_id"**),     inverseJoinColumns=@JoinColumn(name = **"student\_id"**)   )   @JsonBackReference   **private** List<Student> students;    *// Геттеры и сеттеры*    **public** Book() {   } } |

**CRUD в REST**

Рассмотрим методы, часто применяемые при построении REST-сервисов, определенные в спецификации HTTP 1.1:

* **GET** получает информацию о ресурсе по его URI. В контексте REST-сервисов этот метод позволяет запрашивать/получать ресурсы с сервера. Представляет операцию READ (C**R**UD). Возможные статусы ответов: 200(OK), 404(NOT FOUND), 400(BAD REQUEST).
* **POST** используется для передачи серверу объекта сущности, заключенной в запрос, для сохранения его на сервере как нового ресурса. Используется в качестве операции CREATE (**C**RUD). Возможные статусы ответов: 200(OK), 404(NOT FOUND), 400(BAD REQUEST), 201(CREATED).
* **PUT** служит для передачи серверу объекта-сущности и сохранения на стороне сервера под определенным URI. Используется в качестве операции UPDATE (CR**U**D). По спецификации данный метод может создать новый ресурс, если его не существует. С другой стороны, есть возможность оставить задачу создания новых ресурсов на сервере только методу POST. Выбор правила работы остается за разработчиком. Возможные статусы ответов: 200(OK), 400(BAD REQUEST).
* **DELETE** применяется для удаления ресурса. Используется в качестве операции DELETE (CRU**D**).

**Реализация контроллера**

Создадим контроллер для обработки запросов к REST-сервису. Чтобы отделить от основной логики работы веб-сервиса, можно сделать специальный **end-point**, например, **/api/v1**.

|  |
| --- |
| @RequestMapping(**"/api/v1"**) @RestController **public** **class** StudentsRestController {     **private** StudentsService studentsService;      @Autowired     **public** **void** setStudentsService(StudentsService studentsService) {         **this**.studentsService = studentsService;     }      @GetMapping(**"/students/{id}"**)     **public** Student getStudentById(@PathVariable Long id) {         **return** studentsService.getStudentById(id);     }      @GetMapping(**"/students"**)     **public** List<Student> getAllStudents() {         **return** studentsService.getAllStudentsList();     }      @PostMapping(**"/students"**)     **public** Student addStudent(@RequestBody Student student) {         student.setId(0L);         **return** studentsService.saveOrUpdate(student);     }      @PutMapping(path = **"/students"**, consumes = {MediaType.APPLICATION\_JSON\_VALUE})     **public** Student updateStudent(@RequestBody Student student) {         **return** studentsService.saveOrUpdate(student);     }      @DeleteMapping(**"/students/{id}"**)     **public** **int** deleteStudent(@PathVariable Long id) {         studentsService.delete(id);         **return** HttpStatus.OK.value();     }      @ExceptionHandler     **public** ResponseEntity<StudentsErrorResponse> handleException(StudentNotFoundException exc) {         StudentsErrorResponse studentsErrorResponse = **new** StudentsErrorResponse();         studentsErrorResponse.setStatus(HttpStatus.NOT\_FOUND.value());         studentsErrorResponse.setMessage(exc.getMessage());         studentsErrorResponse.setTimestamp(System.currentTimeMillis());         **return** **new** ResponseEntity<>(studentsErrorResponse, HttpStatus.NOT\_FOUND);     } } |

Из приведенного выше кода видно, что в url не включается информация о выполняемом над ресурсами действии. В качестве **end-point** используется один и тот же адрес **/students**, а в зависимости от того, какой именно запрос пришел, выполняется одна из CRUD-операций.

**@RestController** совмещает две аннотации: **@Controller** и **@ResponseBody.** @ResponseBody, добавленный к классу, означает, что над всеми методами этого контроллера будут автоматически проставлены @ResponseBody.

Рассмотрим каждый метод.

* **getStudentById()** — выполняется при получении от клиента GET-запроса **/students/{id}**, в котором в качестве **PathVariable** указан id запрашиваемого студента. Метод через **studentsService** находит в базе данных студента и возвращает его клиенту в виде JSON-объекта.
* **getAllStudents()** — выполняется при получении от клиента GET-запроса **/students** и в качестве ответа возвращает список всех студентов в виде **JSONArray**.
* **addStudent()** — реагирует на POST-запрос **/students**, в который вшит объект типа **Student**. Поскольку **POST** должен отвечать за добавление нового ресурса, id полученного объекта обнуляется.
* **updateStudent()** — получает PUT-запрос **/student**, из которого извлекает объект типа **Student** и использует его для обновления записи в базе данных.
* **deleteStudent()** — выполняется при получении от клиента DELETE-запроса **/students/{id}** и служит для удаления студента из базы данных по его id.

Для обработки возможных исключений реализован метод **handleException()**, который в случае возникновения исключения отправляет ответ клиенту в виде объекта типа **ResponseEntity<StudentsErrorResponse>**. Классы обработки исключений:

|  |
| --- |
| **public** **class** StudentNotFoundException **extends** RuntimeException {     **public** StudentNotFoundException(String message) {         **super**(message);     } }  **public** **class** StudentsErrorResponse {     **private** **int** status;     **private** String message;     **private** **long** timestamp;      // Геттеры и сеттеры      **public** StudentsErrorResponse() {     } } |

**MapStruct**

При разработке веб-приложений зачастую приходится делать преобразования вида Entity => DTO или DTO => Entity. Код для подобного рода операций можно прописывать вручную, но это приведет к появлению большого количества однотипного кода, и следовательно к возможным мелким ошибкам при написании кода. Вместо этого можно взять готовое решение - библиотеку MapStruct, которая на этапе компиляции генерирует код для преобразователей.

Для подключения MapStruct к Maven проекту необходимо:

1. Подключить зависимости

|  |
| --- |
| <dependency>     <groupId>org.mapstruct</groupId>     <artifactId>mapstruct</artifactId>     <version>1.3.1.Final</version>  </dependency>  <dependency>     <groupId>org.mapstruct</groupId>     <artifactId>mapstruct-processor</artifactId>     <version>1.3.1.Final</version>  </dependency> |

1. И в build > plugins добавить

|  |
| --- |
| <plugin>     <groupId>org.apache.maven.plugins</groupId>     <artifactId>maven-compiler-plugin</artifactId>     <version>3.5.1</version>     <configuration>         <source>1.8</source>         <target>1.8</target>         <annotationProcessorPaths>             <path>                 <groupId>org.mapstruct</groupId>                 <artifactId>mapstruct-processor</artifactId>                 <version>1.3.1.Final</version>             </path>         </annotationProcessorPaths>     </configuration>  </plugin> |

1. Готово, можно описывать правила преобразования.

Допустим у нас есть две сущности Category и Item. Item это товар, имеющий id, название, цену и категорию, к которой он относится (смартфоны, ноутбуки, процессоры и пр.). Категория соответственно имеет id, название и список товаров, которые к ней относятся.

|  |
| --- |
| @Entity  @Table(name = **"categories"**)  **public class** Category {     @Id     @GeneratedValue(strategy = GenerationType.***IDENTITY***)     @Column(name = **"id"**)  **private** Long **id**;     @Column(name = **"title"**)  **private** String **title**;     @OneToMany(mappedBy = **"category"**)  **private** List<Item> **items**;     // … геттеры/сеттеры/конструкторы  }  @Entity  @Table(name = **"items"**)  **public class** Item {     @Id     @GeneratedValue(strategy = GenerationType.***IDENTITY***)     @Column(name = **"id"**)  **private** Long **id**;     @Column(name = **"title"**)  **private** String **title**;     @Column(name = **"price"**)  **private int price**;     @ManyToOne     @JoinColumn(name = **"category\_id"**)  **private** Category **category**;     // … геттеры/сеттеры/конструкторы  } |

Давайте начнем с преобразования категории в Data Transfer Object. Для этого создадим класс CategoryDto. В DTO нас интересует только id и название категории и абсолютно не интересует список товаров.

|  |
| --- |
| **public class** CategoryDto {  **private** Long **id**;  **private** String **title**;     // … геттеры/сеттеры/конструкторы  } |

Теперь необходимо объяснить библиотеке MapStruct каким образом должно происходить преобразование Category > CategoryDto и CategorDto > Category. Для этого создается интерфейс, помеченный аннотацией **@Mapper**.

|  |
| --- |
| @Mapper  **public interface** CategoryMapper {     CategoryMapper ***MAPPER*** = Mappers.*getMapper*(CategoryMapper.**class**); // (1)     Category toCategory(CategoryDto categoryDto); // (2)     @InheritInverseConfiguration // (3)     CategoryDto fromCategory(Category category); // (4)     List<Category> toCategoryList(List<CategoryDto> categoryDtos); // (5)     List<CategoryDto> fromCategoryList(List<Category> categories); // (6)  } |

В (1) мы создаем ссылку на mapper для получения к нему доступа в коде проекта (*CategoryMapper.MAPPER.toCategory(...)*). Метод toCategory будет заниматься преобразованием объекта типа CategoryDto в объект типа Category, название метода не имеет значения, mapper будет смотреть на тип аргумента и тип возвращаемого значения. То же самое можно сказать про (3-4), только преобразование производится в обратную сторону. Про @InheritInverseConfiguration будет рассказано на примере Item. Чтобы вручную не заниматься преобразованием листов объектов, добавлены соответствующие методы.

**Каким образом MapStruct понимает как передавать данные из одного типа в другой?** Маппер проводит соответствие через названия полей. То есть значение поля title из Category будет “перенесено” в поле title CategoryDto. И так для всех полей.

Может возникнуть ситуация, при которой названия полей в сущности и дто отличаются. В этом случае поможет аннотация @Mapping. Вот как выглядит маппер для Item.

|  |
| --- |
| @Mapper(uses = { CategoryMapper.**class** })  **public interface** ItemMapper {     ItemMapper ***MAPPER*** = Mappers.*getMapper*(ItemMapper.**class**);     @Mapping(source = **"categoryDto"**, target = **"category"**)     Item toItem(ItemDto itemDto);     List<Item> toItemList(List<ItemDto> itemDtos);     @InheritInverseConfiguration     ItemDto fromItem(Item item);     List<ItemDto> fromItemList(List<Item> items);  } |

Обратите внимание на

|  |
| --- |
| @Mapping(source = **"categoryDto"**, target = **"category"**)  Item toItem(ItemDto itemDto); |

Этот метод занимается преобразованием **ItemDto > Item**, **source = “categoryDto”** означает что в классе источнике (**ItemDto**) есть поле **categoryDto**, которое необходимо преобразовать в поле **category** целевого класса **Item** (**target = “category”**).

Теперь должно стать понятным назначение аннотации **@InheritInverseConfiguration**, мы указываем мапперу, что ему необходимо брать “обратные” правила преобразования из уже описанных выше (условно говоря для **fromItem**: @Mapping(source = **"category"**, target = **"categoryDto"**)). Набор правил может получиться достаточно большим (10+ полей), так что проще сделать вот такую обратную ссылку.

Поскольку ItemDto внутри себя содержит ссылку на CategoryDto, то ему мапперу для Item необходимо уметь работать и с CategoryDto, поэтому мы добавляем ссылку на CategoryMapper.class.

|  |
| --- |
| @Mapper(uses = { CategoryMapper.**class** }) |

После описания всех необходимых мапперов, можем применить их в коде нашего сервера.

|  |
| --- |
| @Service  **public class** ItemService {     // ...  **public** ItemDto save(ItemDto itemDto) {         Item item = ItemMapper.***MAPPER***.toItem(itemDto);         item = **itemRepository**.save(item);  **return** ItemMapper.***MAPPER***.fromItem(item);     }  **public** List<ItemDto> findAll() {  **return** ItemMapper.***MAPPER***.fromItemList(**itemRepository**.findAllItemsWithCategories());     }  **public** ItemDto findOne(Long id) {  **return** ItemMapper.***MAPPER***.fromItem(**itemRepository**.findById(id).get());     }     // ... |

**Практическое задание**

1. Добавить к проекту REST API для одной из сущностей.

**Дополнительные материалы**

1. [Spring REST Docs](https://docs.spring.io/spring-restdocs/docs/2.0.2.RELEASE/reference/html5/).

**Используемая литература**

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>

**Защита приложения**

Spring Security. Авторизация. Защита на уровне запросов, представлений, методов.

**Оглавление**

[Начало работы с Spring Security](https://docs.google.com/document/d/1GqYCiwCBKufCrBaeAsFKt6b7hB1oF_FZjLRgEr0Wz00/edit#heading=h.2et92p0)

[Подготовка базы данных](https://docs.google.com/document/d/1GqYCiwCBKufCrBaeAsFKt6b7hB1oF_FZjLRgEr0Wz00/edit#heading=h.3dy6vkm)

[Авторизация](https://docs.google.com/document/d/1GqYCiwCBKufCrBaeAsFKt6b7hB1oF_FZjLRgEr0Wz00/edit#heading=h.4d34og8)

[Защита на уровне представлений](https://docs.google.com/document/d/1GqYCiwCBKufCrBaeAsFKt6b7hB1oF_FZjLRgEr0Wz00/edit#heading=h.2s8eyo1)

[Защита на уровне методов](https://docs.google.com/document/d/1GqYCiwCBKufCrBaeAsFKt6b7hB1oF_FZjLRgEr0Wz00/edit#heading=h.17dp8vu)

[Практическое задание](https://docs.google.com/document/d/1GqYCiwCBKufCrBaeAsFKt6b7hB1oF_FZjLRgEr0Wz00/edit#heading=h.3rdcrjn)

[Дополнительные материалы](https://docs.google.com/document/d/1GqYCiwCBKufCrBaeAsFKt6b7hB1oF_FZjLRgEr0Wz00/edit#heading=h.lnxbz9)

[Используемая литература](https://docs.google.com/document/d/1GqYCiwCBKufCrBaeAsFKt6b7hB1oF_FZjLRgEr0Wz00/edit#heading=h.35nkun2)

**Начало работы с Spring Security**

Spring Security — это мощный и гибкий фреймворк, предоставляющий механизмы защиты веб-приложения. Предлагает несколько уровней защиты:

* защита на уровне запросов — ограничение доступа к ресурсам, имеющим определенный URL;
* защита на уровне представлений — отображение элементов представления в зависимости от привилегий пользователя;
* защита на уровне методов — ограничение вызова методов сервисов/контроллеров в зависимости от привилегий пользователя.

Для подключения Spring Security к проекту, в pom.xml необходимо добавить следующие зависимости:

|  |
| --- |
| <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-security</artifactId>  </dependency>  <dependency>      <groupId>org.thymeleaf.extras</groupId>      <artifactId>thymeleaf-extras-springsecurity5</artifactId>      <version>3.0.4.RELEASE</version>  </dependency> |

Первая зависимость представляет собой сам модуль Spring Security, а вторая — библиотеку для его интеграции с шаблонизатором Thymeleaf. После того, как модуль Spring Security был подключен, Spring Boot автоматически выполнит базовую настройку правила безопасности. Как правило, такой настройки недостаточно, поэтому рассмотрим “ручное” конфигурирование этих правил.

Для начала добавим в проект конфигурационный класс SecurityConfig, наследуемый от WebSecurityConfigurerAdapter. 

|  |
| --- |
| @Configuration @EnableWebSecurity @EnableGlobalMethodSecurity(securedEnabled = **true**) **public** **class** SecurityConfig **extends** WebSecurityConfigurerAdapter {    *// ...* } |

Аннотация @Configuration говорит о том, что данный класс является конфигурационным. @EnableWebSecurity отключает стандартные настройки безопасности Spring Security и начинает использовать правила, прописанные в SecurityConfig. @EnableGlobalMethodSecurity активирует возможность ставить защиту на уровне методов (для этого над методами ставятся аннотации @Secured и @PreAuthorized).

Вот так может выглядеть полноценная настройка безопасности приложения.

|  |
| --- |
| @Configuration @EnableWebSecurity @EnableGlobalMethodSecurity(prePostEnabled = **true**) **public** **class** SecurityConfig **extends** WebSecurityConfigurerAdapter {     **private** DataSource dataSource;      @Autowired     **public** **void** setDataSource(DataSource dataSource) {         **this**.dataSource = dataSource;     }      @Override     **protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {         auth.jdbcAuthentication().dataSource(dataSource);     }          @Override     **protected** **void** configure(HttpSecurity http) **throws** Exception {         http.authorizeRequests()                 .antMatchers(**"/"**).hasAnyRole(**"USER"**)                 .antMatchers(**"/admin/\*\*"**).hasRole(**"ADMIN"**)                   .and()                 .formLogin()                   .loginPage(**"/login"**)                   .loginProcessingUrl(**"/authenticateTheUser"**)                 .permitAll();     } } |

За настройку способа аутентификации отвечает метод configure(AuthenticationManagerBuilder auth). Существует несколько стандартных вариантов настройки этого метода:

1. Spring Security при проверке логина/пароля будет искать в стандартной паре таблиц: users и authorities. Все что нужно для такого способа, это заинжектить DataSource, создать таблицы в базе, и добавить туда пользователей. Настройка самого метода будет выглядеть вот так:

|  |
| --- |
| @Override **protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {     auth.jdbcAuthentication().dataSource(dataSource); } |

Бин DataSource настраивается через Spring Boot application.properties:

|  |
| --- |
| spring.datasource.driver-class-name=org.postgresql.Driver  spring.datasource.url=jdbc:postgresql://localhost:5432/postgres?currentSchema=spring\_app  spring.datasource.username=postgres  spring.datasource.password=iejfJ1Kw |

1. В некоторых случаях нет необходимости в создании новых пользователей и хранении их в базе данных. Например, мы можем хранить просто в памяти стандартный набор пользователей и работать с ними (при этом отпадает необходимость в бине DataSource). В этом случае подойдет вариант с inMemoryAuthentication():

|  |
| --- |
| @Override  **protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {   User.UserBuilder users = User.withDefaultPasswordEncoder();   auth.inMemoryAuthentication()    .withUser(users.username(**"user1"**).password(**"pass1"**).roles(**"USER"**, **"ADMIN"**))    .withUser(users.username(**"user2"**).password(**"pass2"**).roles(**"USER"**)); } |

User.UserBuilder users служит для создания пользователей (их логина, пароля и ролей). Метод auth.inMemoryAuthentication() создает пользователей и указывает необходимость брать информацию о них из памяти.

1. Если же мы хотим хранить пользователей и их роли в базе данных, и у нас есть собственные таблицы для этого, то можно воспользоваться бином DaoAuthenticationProvider (при таком подходе отпадает необходимость в методе configure(AuthenticationBuilder)). Бин UserService implements UserDetailsService описывает как должна производиться загрузка пользовательских данных из базы, чуть ниже будет приведен пример с реализацией. BCryptPasswordEncoder указывает что пароли должны храниться не в “чистом” виде и хэшироваться с помощью BCrypt.

|  |
| --- |
| @Configuration  @EnableWebSecurity  @EnableGlobalMethodSecurity(securedEnabled = **true**)  **public class** SecurityConfig **extends** WebSecurityConfigurerAdapter {  **private** UserService **userService**;     @Autowired  **public void** setUserService(UserService userService) {  **this**.**userService** = userService;     }     @Bean  **public** BCryptPasswordEncoder passwordEncoder() {  **return new** BCryptPasswordEncoder();     }     @Bean  **public** DaoAuthenticationProvider authenticationProvider() {         DaoAuthenticationProvider auth = **new** DaoAuthenticationProvider();         auth.setUserDetailsService(**userService**);         auth.setPasswordEncoder(passwordEncoder());  **return** auth;     }  } |

Давайте посмотрим на пример реализации UserService. В данном примере User и Role это самые обычные сущности. Для того, чтобы достать их из базы используем Spring Data репозитории, поэтому Security Config не требует явного указания бина DataSource.

|  |
| --- |
| @Service  **public class** UserService **implements** UserDetailsService {  **private** UserRepository **userRepository**;  **private** RoleRepository **roleRepository**;     @Autowired  **public void** setUserRepository(UserRepository userRepository) {  **this**.**userRepository** = userRepository;     }     @Autowired  **public void** setRoleRepository(RoleRepository roleRepository) {  **this**.**roleRepository** = roleRepository;     }  **public** User findByUsername(String username) {  **return userRepository**.findOneByUsername(username);     }     @Override  **public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {         User user = **userRepository**.findOneByUsername(username);  **if** (user == **null**) {  **throw new** UsernameNotFoundException(**"Invalid username or password"**);         }  **return new** org.springframework.security.core.userdetails.User(user.getPhone(), user.getPassword(), mapRolesToAuthorities(user.getRoles()));     }  **private** Collection<? **extends** GrantedAuthority> mapRolesToAuthorities(Collection<Role> roles) {  **return** roles.stream().map(role -> **new** SimpleGrantedAuthority(role.getName())).collect(Collectors.*toList*());     }  } |

С правилами аутентификации разобрались. Переходим непосредственно к защите приложения. За настройку безопасности отвечает метод configure(HttpSecurity http). В нем можно выполнить настройку процесса логина и логаута, правил доступа к определенным частям веб-приложения, и т.д. Пример настройки представлен ниже.

|  |
| --- |
| @Configuration @EnableWebSecurity @EnableGlobalMethodSecurity(prePostEnabled = **true**) **public** **class** SecurityConfig **extends** WebSecurityConfigurerAdapter {     // ...         @Override     **protected** **void** configure(HttpSecurity http) **throws** Exception {         http.authorizeRequests()                 .antMatchers(**"/"**).hasAnyRole(**"USER"**)                 .antMatchers(**"/admin/\*\*"**).hasRole(**"ADMIN"**)                   .and()                 .formLogin()                   .loginPage(**"/login"**)                   .loginProcessingUrl(**"/authenticateTheUser"**)                 .permitAll();     } } |

* Метод **configure(HttpSecurity http)** отвечает за настройку защиты на уровне запросов и конфигурирование процессов авторизации.
* **antMatchers** — с помощью данного метода указывается http-метод и URL (или шаблон URL), доступ к которому необходимо ограничить.
* **hasRole(String role)**, **hasAnyRole(String... roles)** — в нем указывается одна роль или набор ролей, необходимых пользователю для доступа к данному ресурсу.
* **formLogin()** — дает возможность настроить форму для авторизации.
* **loginPage** — URL формы авторизации.
* **loginProcessingUrl** — URL, на который будут отправляться данные формы (методом POST).
* **\* logout()** — позволяет настроить правила выхода из учетной записи.
* **\* failureUrl** — адрес для перенаправления пользователя в случае неудачной авторизации.
* **\* logoutSuccessUrl** — URL, на который будет перенаправлен пользователь при выходе из аккаунта автора.
* **\* usernameParameter** и **passwordParameter** — имена полей формы, содержащие логин и пароль, если не используются стандартные имена username и password;

(\*) — параметры, не использующиеся в примере.

В примере выше **configure(HttpSecurity http)** указывает, что для доступа к сайту пользователь должен быть обязательно авторизован и иметь роль USER, иначе он будет перенаправлен на страницу авторизации. Для доступа к запросам, начинающимся с **/admin/**, необходимо иметь право доступа ADMIN. Для авторизации используется форма, для доступа к которой необходимо обратиться по адресу **/login**. Результаты заполнения этой формы в виде POST-запроса будут отправлены на **URL /authentificateTheUser**.

**Подготовка базы данных**

Рассмотрим работу через jdbcAuthentication. Чтобы хранить информацию о пользователях и работать с ней, применим MySQL базу данных. По умолчанию Spring Security будет использовать стандартный шаблон таблиц в БД: пользователи хранятся в таблице **users**, а роли — в **authorities**.

В таблице **users** три столбца:

* **username** — имя пользователя;
* **password** — пароль, который может храниться как в открытом, так и в хешированном виде;
* **enabled** — возможность пользователя войти под данной учетной записью.

Запрос на создание такой таблицы и добавление тестовых пользователей:

|  |
| --- |
| CREATE TABLE users (     username varchar(50) NOT NULL,     password varchar(100) NOT NULL,     enabled tinyint(1) NOT NULL,      PRIMARY KEY (username) ) ENGINE=InnoDB DEFAULT CHARSET=latin1;  INSERT INTO users VALUES ('user1', '{noop}123', 1), ('user2', '{noop}123', 1); |

Таблица **authorities** включает имя пользователя и соответствующую ему роль — каждому можно добавить по несколько. Запрос на создание этой таблицы и добавление тестовых данных:

|  |
| --- |
| CREATE TABLE authorities (     username varchar(50) NOT NULL,     authority varchar(50) NOT NULL,      UNIQUE KEY authorities\_idx\_1 (username, authority),      CONSTRAINT authorities\_ibfk\_1     FOREIGN KEY (username)     REFERENCES users (username) ) ENGINE=InnoDB DEFAULT CHARSET=latin1;  INSERT INTO authorities VALUES ('user1', 'ROLE\_ADMIN'), ('user1', 'ROLE\_USER'), ('user2', 'ROLE\_USER'); |

Выполнив эти два скрипта, мы подготовили БД для использования в качестве источника данных — при авторизации и указании прав при использовании веб-приложения.

Авторизация

Настройка базы данных, Spring Security и пользователей выполнена. Теперь реализуем возможность авторизации на сайте. Необходимы форма и метод обработки GET-запроса для нее.

|  |
| --- |
| @GetMapping(**"/login"**) **public** String showMyLoginPage() {     **return** **"modern-login"**; } |

Метод очень простой — всего лишь возвращает html-страницу с именем **modern-login.html**. Код страницы:

|  |
| --- |
| <!DOCTYPE html>  <html xmlns:th="http://www.thymeleaf.org" lang="en">  <head>     <title>Login Page</title>     <meta charset="utf-8">     <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">     <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">     <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js"></script>     <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script> </head>  <body>  <div>     <div id="loginbox" style="margin-top: 50px;" class="mainbox col-md-3 col-md-offset-2 col-sm-6 col-sm-offset-2">         <div class="panel panel-info">             <div class="panel-heading">                 <div class="panel-title">Sign In</div>             </div>             <div style="padding-top: 30px" class="panel-body">                 <form th:action="@{/authenticateTheUser}" method="POST" class="form-horizontal">                     <div class="form-group">                         <div class="col-xs-15">                             <div>                                 <div th:if="${param.error} != null">                                     <div class="alert alert-danger col-xs-offset-1 col-xs-10">                                         Invalid username or password                                     </div>                                 </div>                                 <div th:if="${param.logout} != null">                                     <div class="alert alert-success col-xs-offset-1 col-xs-10">                                         You have been logged out.                                     </div>                                 </div>                             </div>                         </div>                     </div>                     <div style="margin-bottom: 25px" class="input-group">                         <span class="input-group-addon"><i class="glyphicon glyphicon-user"></i></span>                         <input type="text" name="username" placeholder="username" class="form-control">                     </div>                     <div style="margin-bottom: 25px" class="input-group">                         <span class="input-group-addon"><i class="glyphicon glyphicon-lock"></i></span>                         <input type="password" name="password" placeholder="password" class="form-control">                     </div>                     <div style="margin-top: 10px" class="form-group">                         <div class="col-sm-6 controls">                             <button type="submit" class="btn btn-success">Login</button>                         </div>                     </div>                 </form>             </div>         </div>     </div> </div> </body> </html> |

У полей для ввода логина и пароля стандартные имена: **username** и **password**. Форма посылает POST-запрос по адресу **@{/authentificateTheUser}**. Если мы попали на эту страницу после неудачной попытки авторизации или после выхода из учетной записи, на форме будут показаны соответствующие сообщения. После авторизации пользователь получает набор прав, указанный в БД, и может пользоваться веб-приложением.

Защита на уровне представлений

Рассмотрим, как изменять видимость элементов на странице в зависимости от прав пользователей. Для этого используется зависимость **thymeleaf-extras-springsecurity5** из **pom.xml**.  
Пример использования таких дополнительных возможностей:

|  |
| --- |
| <!DOCTYPE html> <html xmlns:th="http://www.thymeleaf.org"> <html xmlns:sec="http://www.thymeleaf.org/extras/spring-security"> *<!--...-->* <body> <div class="container">     <h1>Welcome page</h1>     <div sec:authorize="isAuthenticated()">         Authenticated username:         <div sec:authentication="principal.username"></div>         Authenticated user roles:         <div sec:authentication="principal.authorities"></div>     </div>      *<!--<div sec:authorize="hasAnyRole('ADMIN', 'USER')">-->*     <div sec:authorize="hasRole('ADMIN')">         This content will only be visible to ADMIN users.     </div>      <h2>Index:</h2> *<!--...-->* </div> </body> </html> |

Здесь **<div sec:authorize="isAuthenticated()">** отвечает за проверку авторизации пользователя. Если он авторизован, на странице отображаются его **username** и права доступа (например, **ROLE\_ADMIN** или **ROLE\_USER**). Методы **hasRole()** и **hasAnyRole()** проверяют у пользователя наличие определенной роли.

Защита на уровне методов

Последний шаг для создания надежной защиты приложения — ограничить доступ на уровне методов. Для этого можно использовать аннотацию **@Secured**, которая ограничивает доступ к отдельным методам на основе информации о правах текущего пользователя. Пример кода:

|  |
| --- |
| @Secured({**"ROLE\_ADMIN"**}) @RequestMapping(**"/onlyYou"**) @ResponseBody **public** String pageOnlyForAdmins() {     **return** **"index"**; } |

Доступ к методу pageOnlyForAdmins() имеют только пользователи с ролью ADMIN.

**Практическое задание**

1. Создать страницу со списком товаров, на которой можно добавлять позиции и редактировать существующие. На эту страницу должны иметь доступ админы и менеджеры.
2. Создать страницу со списком всех пользователей, к которой имеют доступ только админы.
3. \* Добавить роль суперадмина и дать ему возможность создавать новых пользователей и указывать роли существующим.

**Дополнительные материалы**

1. Вебинар “Как устроен Spring Security”: <https://geekbrains.ru/events/2176>
2. [Обзор Spring Security](https://habrahabr.ru/post/203318/).
3. [Хеширование паролей в Spring Security](https://www.mkyong.com/spring-security/spring-security-password-hashing-example).

**Используемая литература**

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Крис Шефер, Кларенс Хо. Spring 4 для профессионалов (4-е издание).
2. Крейг Уоллс. Spring в действии.