GeekBrains

Основы веб-разработки на Spring Framework

Spring MVC

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

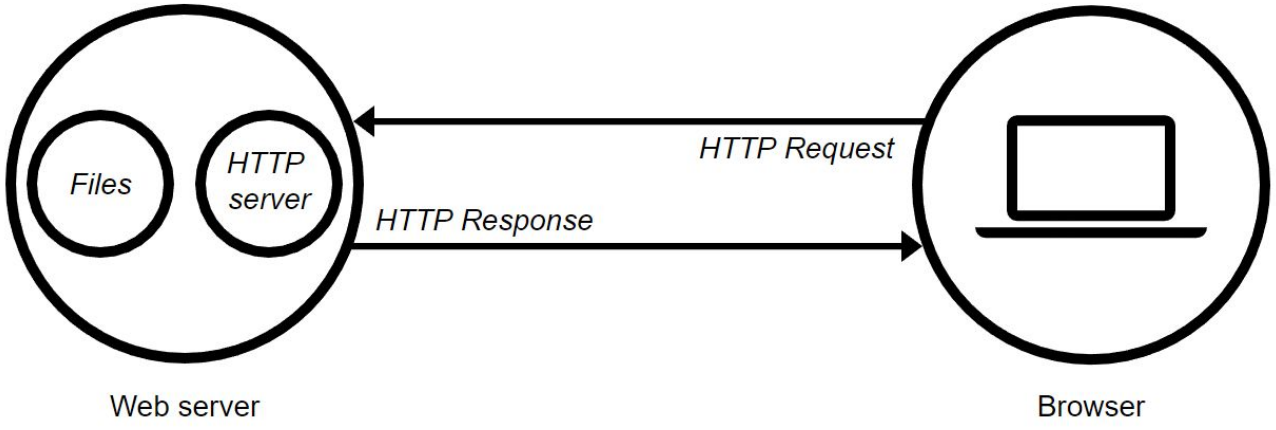
Оглавление

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 (веб-адреса) и 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) 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.

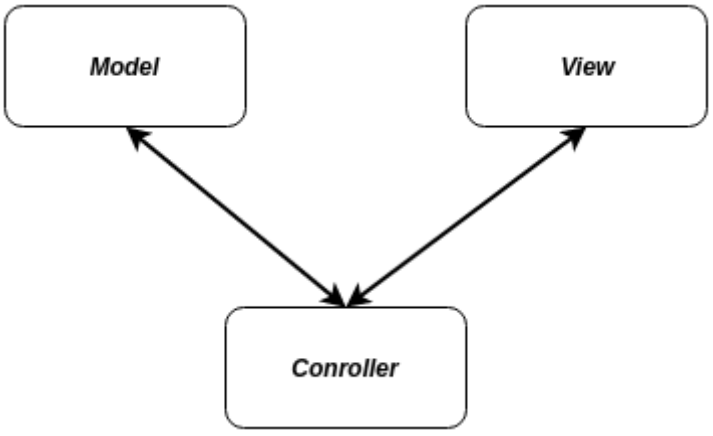
Если же попросить веб-сервер выполнить 1 + 1, то он не сможет этого сделать. Для выполнения на стороне сервера какой-либо логики необходимо передать входящий запрос нашему серверу приложения.

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.

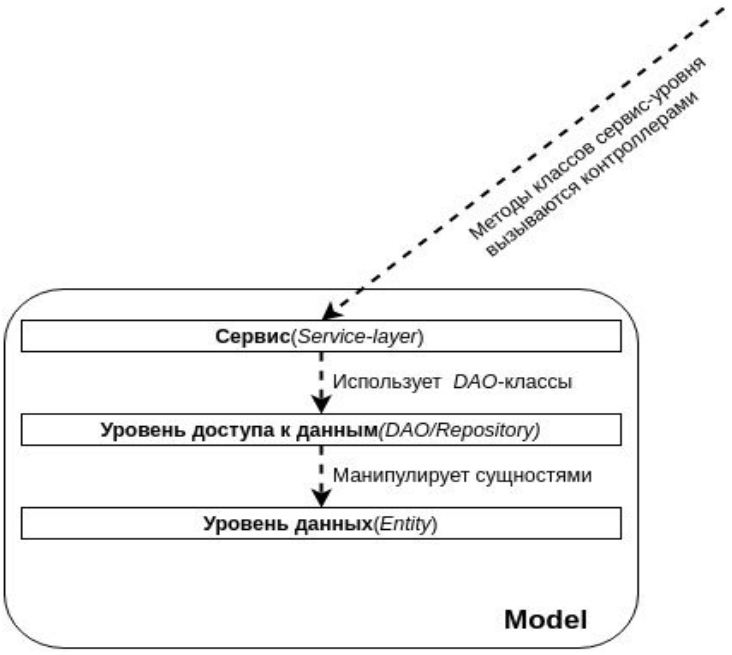
Внутри сервера приложения запрос передается определенному сервлету. Обработка запроса сервлетом производится в отдельном потоке. Сервлет представляет собой Java веб-компонент, управляемый контейнером, и способный предоставлять динамический контент. Аналогично и другим Java компонентам, сервлеты представляют 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):

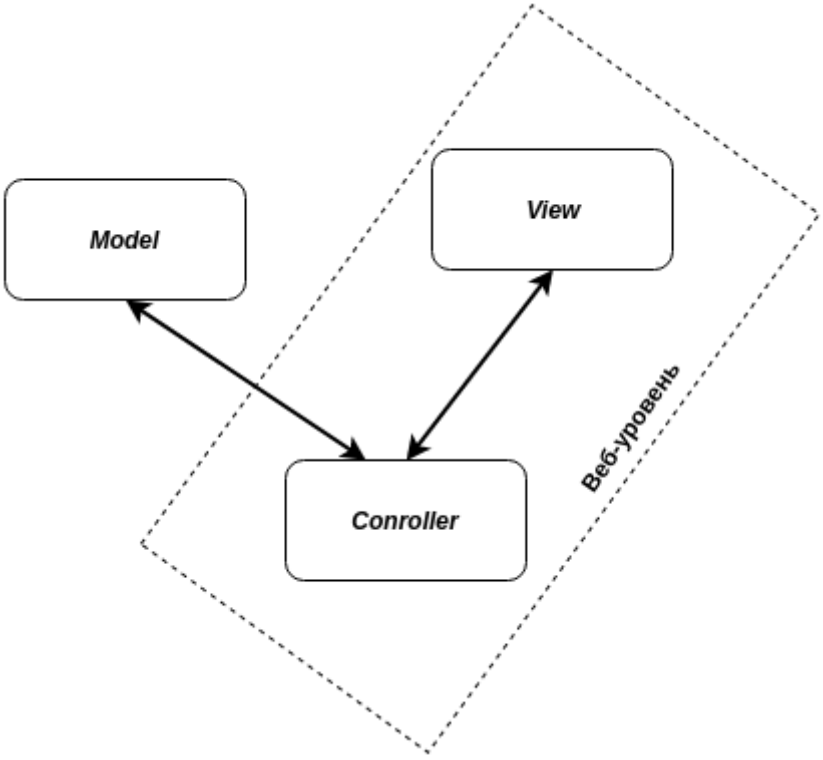


Элемент Model представляет собой данные (класс-сущности и уровень доступа к ним) и механизмы манипуляции этими данными (сервис-уровень):



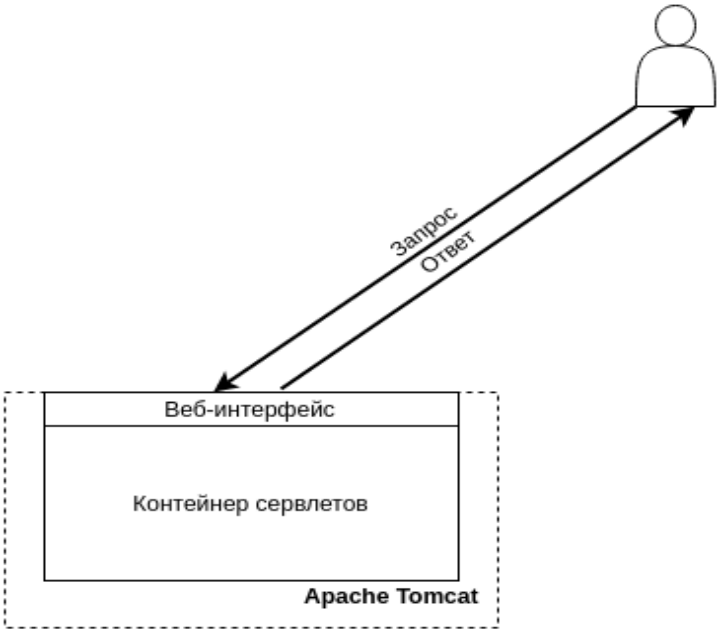
Ранее мы сами непосредственно в классе **Main** инициализировали контекст, получали бин класса сервис-уровня и вызывали его методы. В MVC-архитектуре контекст будет инициализироваться автоматически при разворачивании приложения на веб-сервере, а использовать методы классов сервис-уровня будет контроллер.

Рассмотрим организацию веб-уровня:



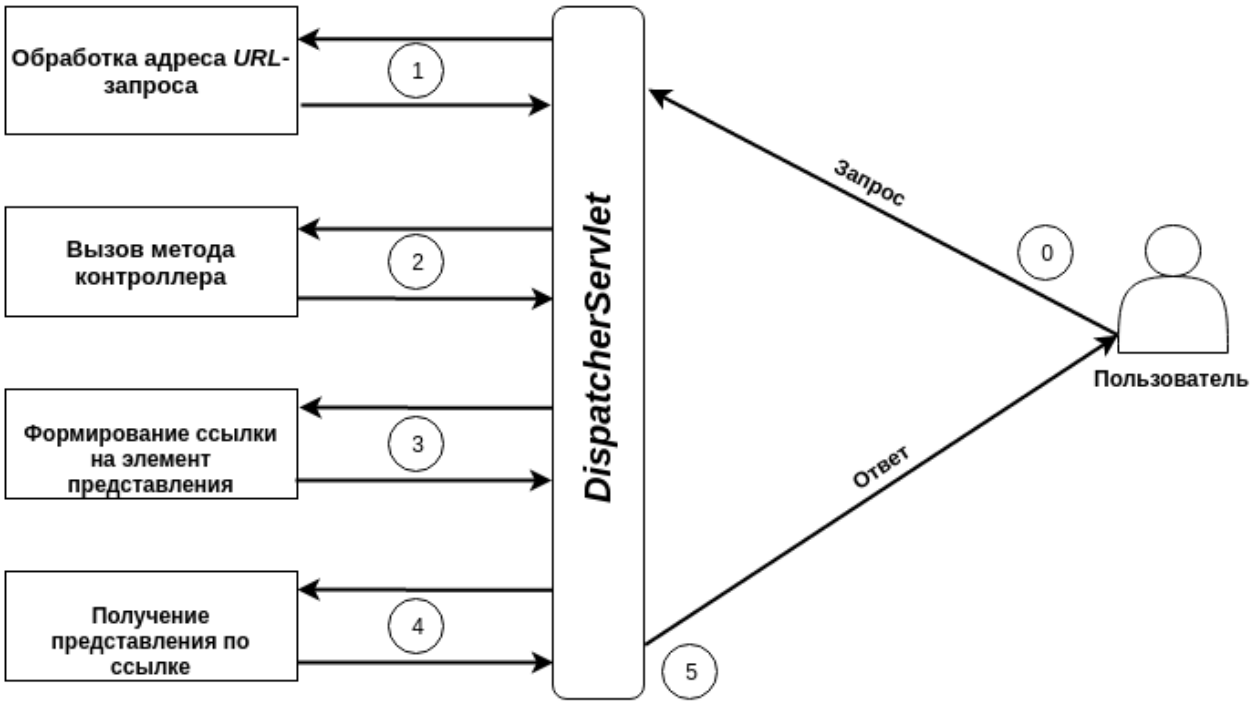
Конечная цель разработки веб-уровня — возможность взаимодействия пользователей с приложением непосредственно через свой браузер. Но кроме непосредственной реализации веб-уровня необходимо разработать соответствующую инфраструктуру, чтобы любой пользователь мог взаимодействовать с приложением через http/https-протокол.

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 **DispatcherServlet**, 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 с 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:

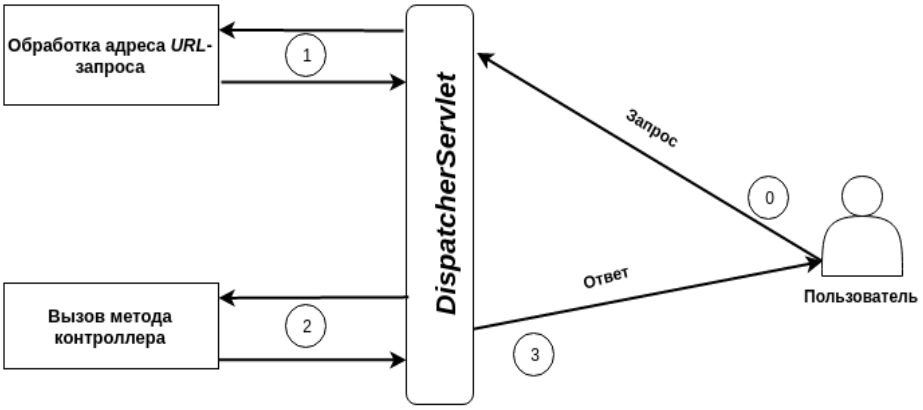
```
@Controller
@RequestMapping("/authors")
public class AuthorsController {
  private AuthorsService authorsService;

  @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 **DispatcherServlet**, 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) {
```

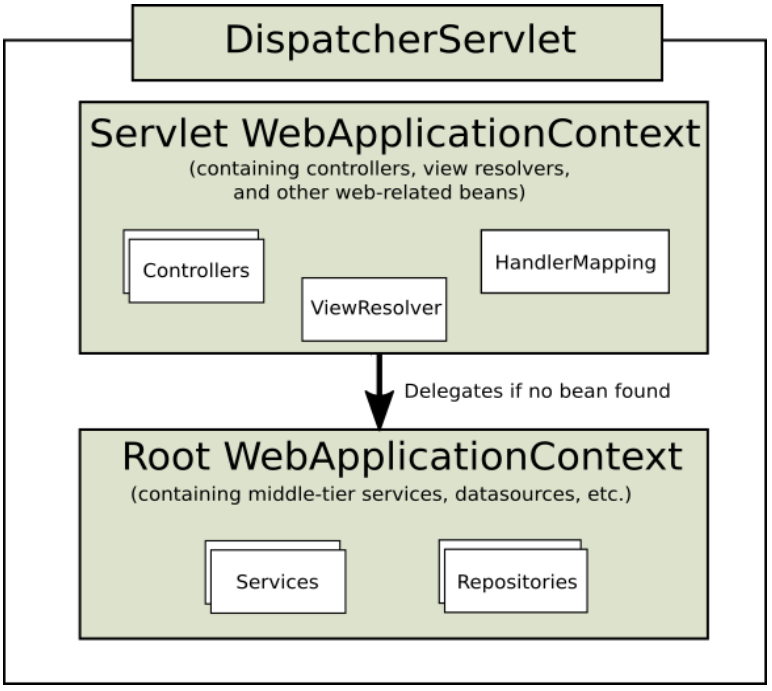
```
    this.authorsService = authorsService;
  }

  @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-запросом. 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 взаимодействия 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**. 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 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** 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]/users/profile?id=1
[http://localhost:8189/application]/video?id=13432&category=edu
```

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**, и на местах чисел 12 и 10 могут попадаться различные значения, в таком случае мы можем получать эти значения с помощью @PathVariable.

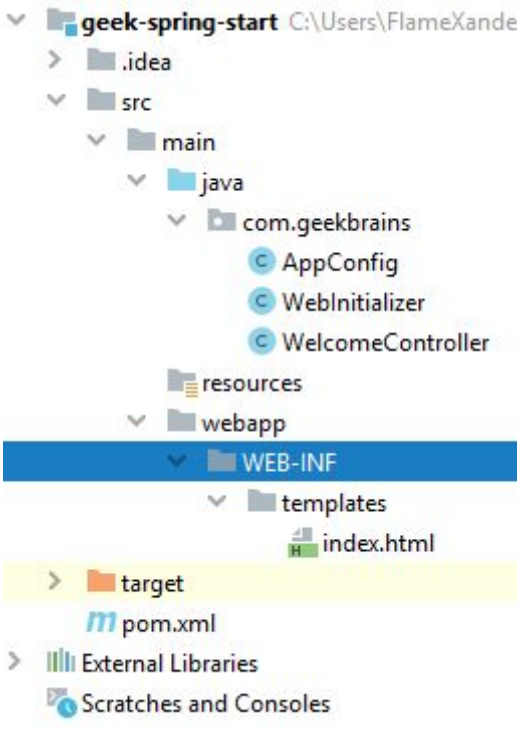
```
@GetMapping(value = "/user/{user_id}/course/{course_id}")
public String pathDemo(@PathVariable(name = "user_id") Long userId,
@PathVariable(name = "course_id") Long courseId) {
   // ... тут какая-то логика ...
}
```

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:

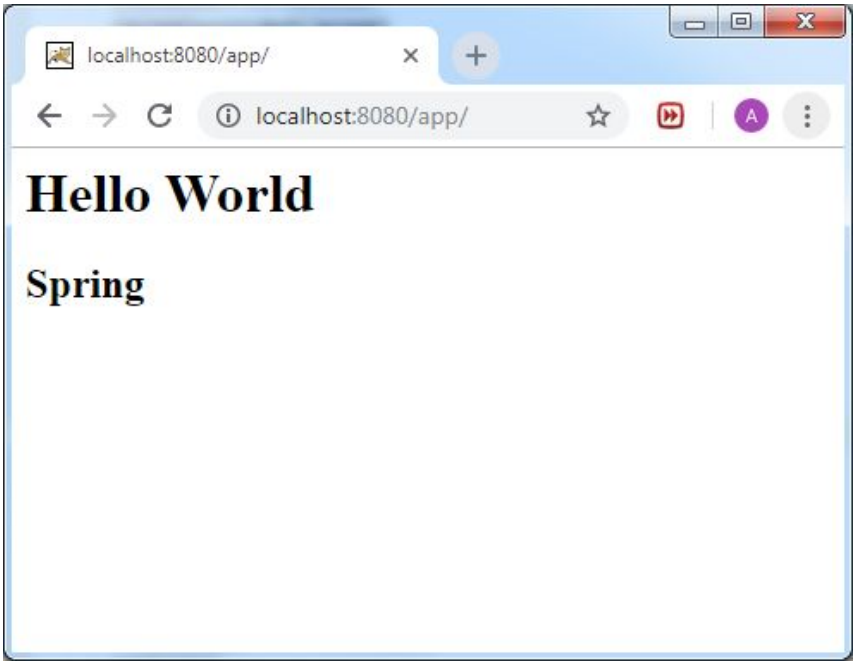
```
<tomcat-users xmlns="http://tomcat.apache.org/xml"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://tomcat.apache.org/xml tomcat-users.xsd"
              version="1.0">
<role rolename="manager-gui"/>
<role rolename="manager-script"/>
<user 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

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