

1. Spring. Первое приложение

Spring Boot, Core, MVC, controller, configuration, logging

1. Настроим проект Spring с использованием <http://start.spring.io>

Spring Initializr позволяет "набрать" в свое приложение нужных компонентов, которые потом Spring Boot (он автоматически включен во все проекты, созданные на Initializr) соберет воедино.

Создадим пример веб-приложения, которое отдает welcome страницу, обращается к собственному API, получает данные и выводит их в таблицу.

Как показано на рисунке, необходимо выполнить следующие шаги: Идем на start.spring.io и создаем проект с зависимостями *Web*, *DevTools*, *JPA* (доступ к реляционным базам), *H2* (простая in-memory база), *Thymeleaf* (движок шаблонов).

Thymeleaf является Java XML/XHTML/HTML5 Template Engine который может работать со средой Web и не Web средой. Он больше подходит при использовании для сервиса **XHTML/HTML5** на уровне **View** (View Layer) приложения **Web** основываясь на структуре **MVC**. Может обрабатывать любой файл XML, даже среды offline (оффлайн). Поддерживается полностью с **Spring Framework**.

Thymeleaf можно использовать, чтобы заменить JSP на уровне View (View Layer) приложения Web MVC. Thymeleaf является программным обеспечением с открытым исходным кодом, с лицензией Apache 2.0.

Thymeleaf Template является шаблонным файлом. В шаблонных файлах (Template file) имеются **Thymeleaf Marker** (Отметки Thymeleaf). Thymeleaf Engine анализирует шаблонный файл (Template file), и сочетается с данными Java, чтобы генерировать новый документ.



Project

☒ Maven Project ☐ Gradle Project

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 2.4.0 (SNAPSHOT) ☐ 2.4.0 (M2) ☐ 2.3.4 (SNAPSHOT) ☒ 2.3.3
☐ 2.2.10 (SNAPSHOT) ☐ 2.2.9 ☐ 2.1.17 (SNAPSHOT) ☐ 2.1.16

Project Metadata

Group

Artifact

Name

Description

Package name

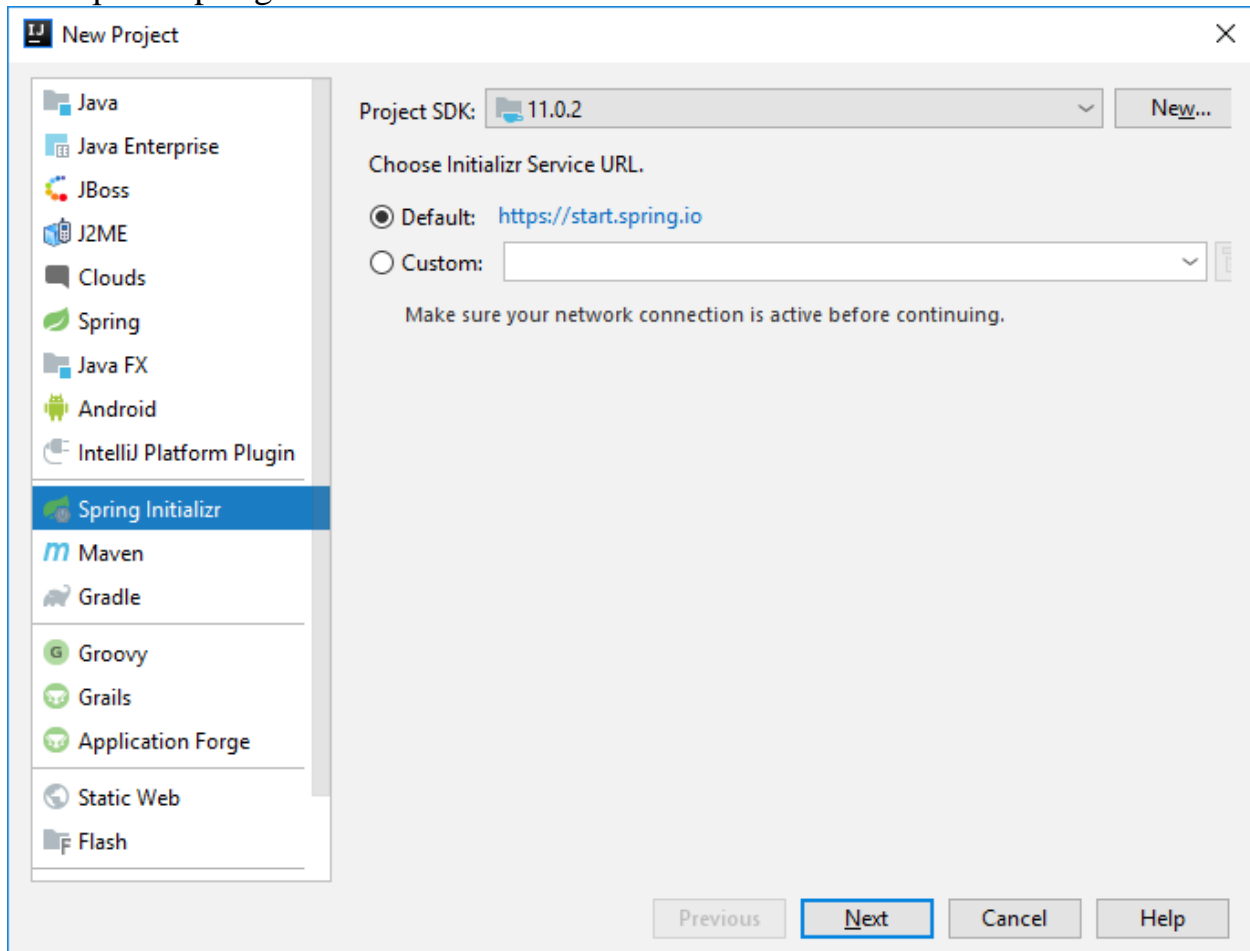
Packaging ☐ Jar ☒ War

Java ☐ 14 ☒ 11 ☐ 8

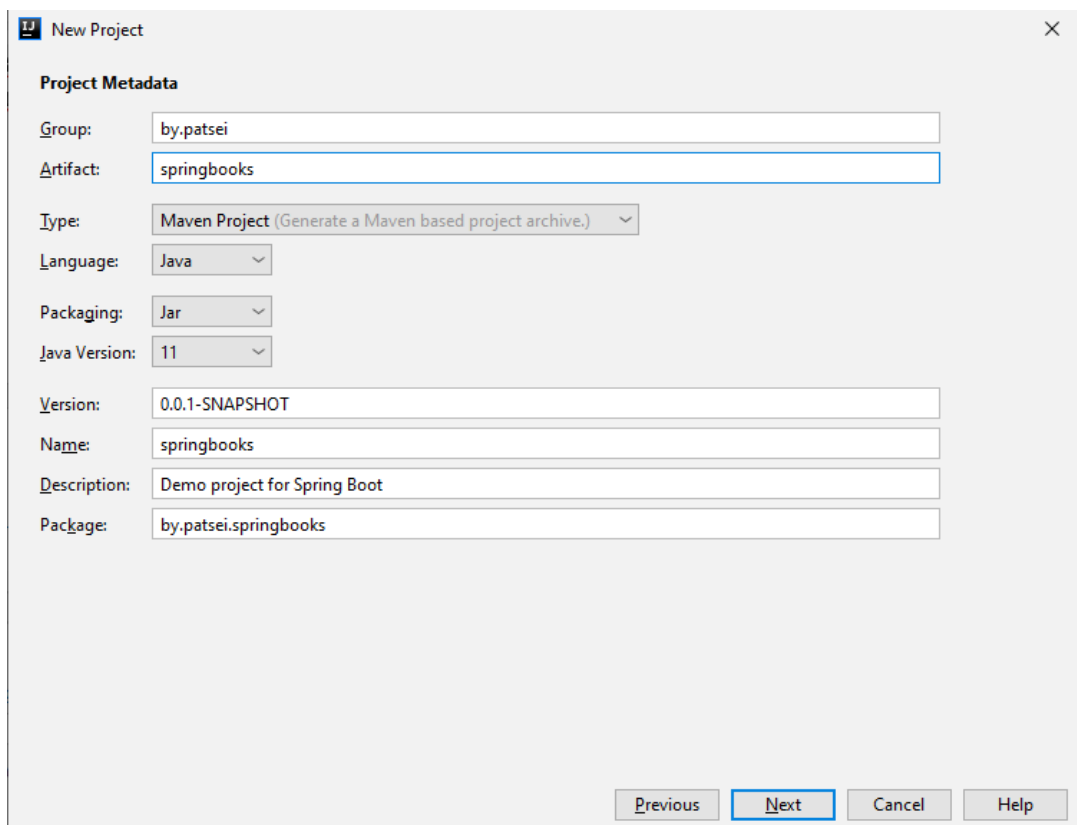
Нажмите Создать проект. Импортируйте проект в IntelliJIdea.

Второй способ можно это же сделать при создании проекта в IntelliJIdea.

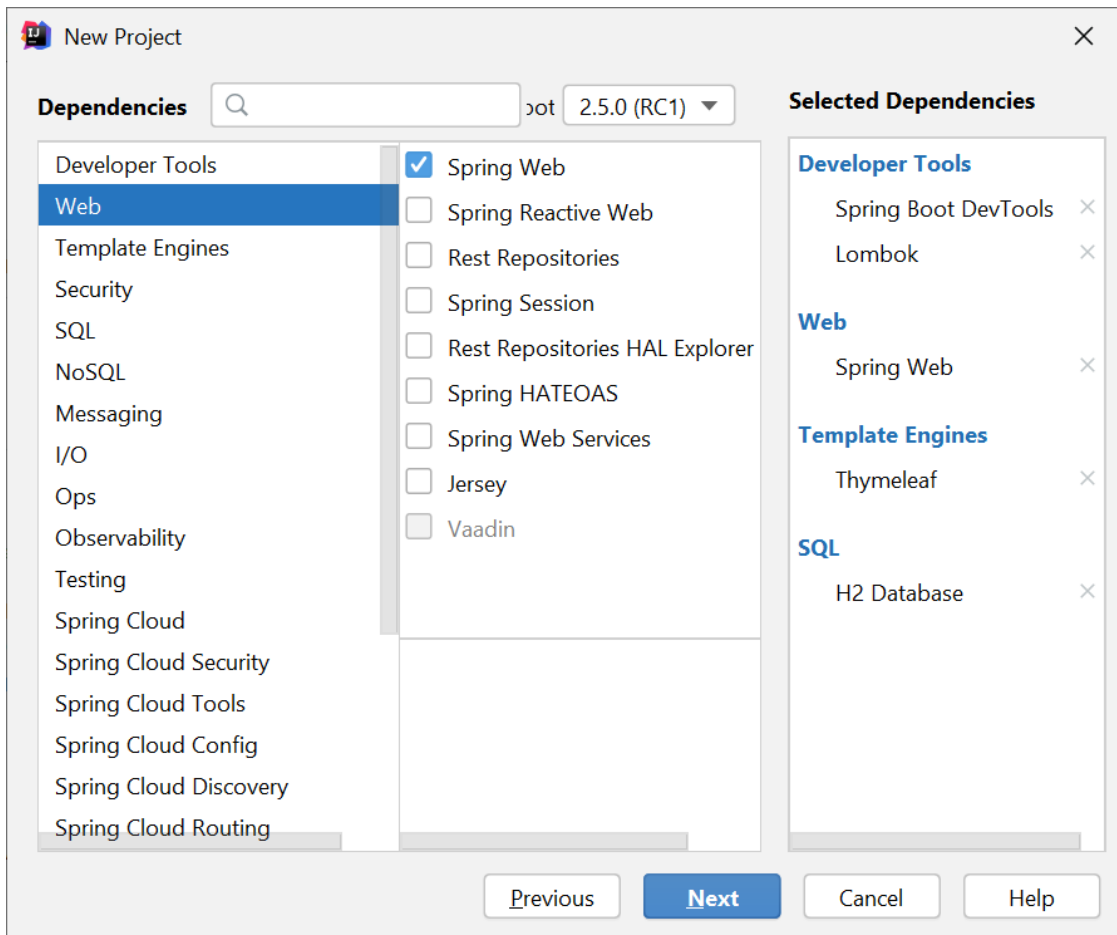
Выберите SpringInitializer



The image shows the 'New Project' dialog in IntelliJ IDEA. On the left, a list of project types is shown, with 'Spring Initializr' selected and highlighted in blue. Other options include Java, Java Enterprise, JBoss, J2ME, Clouds, Spring, Java FX, Android, IntelliJ Platform Plugin, Maven, Gradle, Groovy, Grails, Application Forge, Static Web, and Flash. On the right, the 'Project SDK' is set to '11.0.2'. Below that, the 'Choose Initializr Service URL' section has 'Default' selected with the URL 'https://start.spring.io'. A 'Custom' option is also available with an empty text field. A note states: 'Make sure your network connection is active before continuing.' At the bottom, there are four buttons: 'Previous', 'Next' (which is highlighted with a blue border), 'Cancel', and 'Help'.



The image shows the 'New Project' dialog in IntelliJ IDEA, specifically the 'Project Metadata' screen. It contains several input fields and dropdown menus for configuring the project. The fields are: 'Group' (by.patsei), 'Artifact' (springbooks), 'Type' (Maven Project (Generate a Maven based project archive.)), 'Language' (Java), 'Packaging' (Jar), 'Java Version' (11), 'Version' (0.0.1-SNAPSHOT), 'Name' (springbooks), 'Description' (Demo project for Spring Boot), and 'Package' (by.patsei.springbooks). At the bottom, there are four buttons: 'Previous', 'Next' (highlighted with a blue border), 'Cancel', and 'Help'.



Запустите проект. Изучите структуру проекта.

Ваше приложение начинается выполнением класса **SpringbooksApplication**.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringbooksApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbooksApplication.class, args);
    }

}
```

Этот класс аннотирован через **@SpringBootApplication**. Он выполняет автоматическую конфигурацию Spring, и автоматически сканирует (scan) весь проект, чтобы найти компоненты Spring (Controller, Bean, Service, ...)

Spring Boot

@SpringBootApplication

=

Traditional spring

@Configuration

+

@EnableAutoConfiguration

+

@ComponentScan

После запуска проекта введите в браузере localhost:8080. Так как проект пустой, то вы увидите следующее:

← → ↻ 🏠 ⓘ localhost:8080

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Mon Aug 31 14:21:43 MSK 2020

There was an unexpected error (type=Not Found, status=404).

No message available

Для статических ресурсов (Static Resource), например файлов **css, javascript, image**,... существует папка **src/main/resources/static** и размещать можно в папках. Добавьте в папку static папку css и файл со стилями - style.css.

▼ 📁 resources
▼ 📁 static.css
📄 style.css
> 📁 templates

Запустите проект и обратитесь следующим образом:

```
localhost:8080/css/style.css

@charset "UTF-8";
@import url(https://fonts.googleapis.com/css?family=Open+Sans:300,400,700);

body, input, button{
    font-family: 'Open Sans', sans-serif;
    font-weight: 300;
    line-height: 1.42em;
    color:#A7A1AE;
    background-color:#1F2739;
}

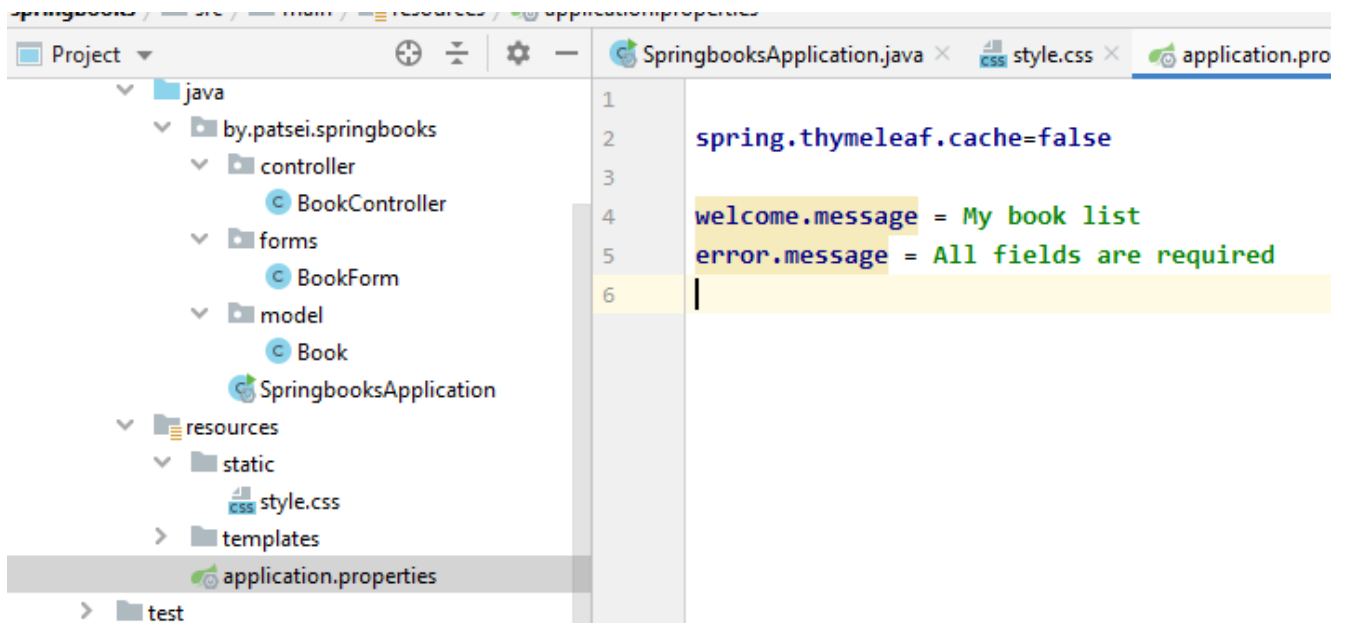
h1 {
    font-size:3em;
    font-weight: 300;
    line-height:1em;
    text-align: center;
    color: #4DC3FA;
}

h2 {
    font-size:1em;
    font-weight: 300;
    text-align: center;
    display: block;
    line-height:1em;
    padding-bottom: 2em;
    color: #FB667A;
}

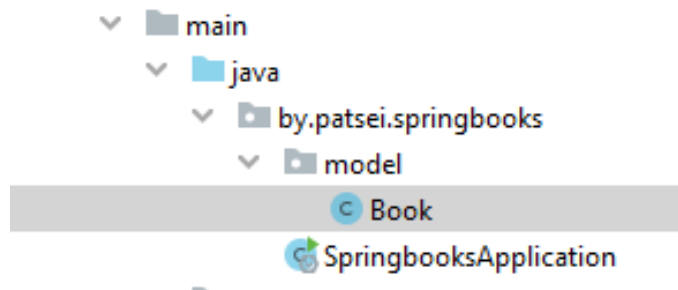
h2 a {
    font-weight: 700;
    text-transform: uppercase;
    color: #FB667A;
    text-decoration: none;
```

2. Создание Model, View и определение Controller

Допишите в файл `application.properties` несколько строк с сообщениями, как показано на рисунке ниже.



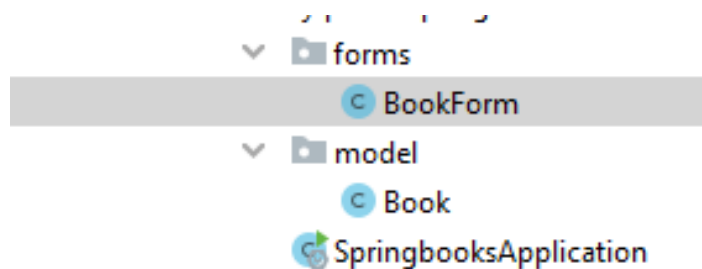
Создадим модель. Это будет класс **Book** с двумя полями



```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Book {
    private String title;
    private String author;
}
```

Класс **BookForm** будет представлять данные **FORM** когда создается новый **Book** на странице.



```
import lombok.AllArgsConstructor;
import lombok.Data;
```

```
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class BookForm {
    private String title;
    private String author;
}
```

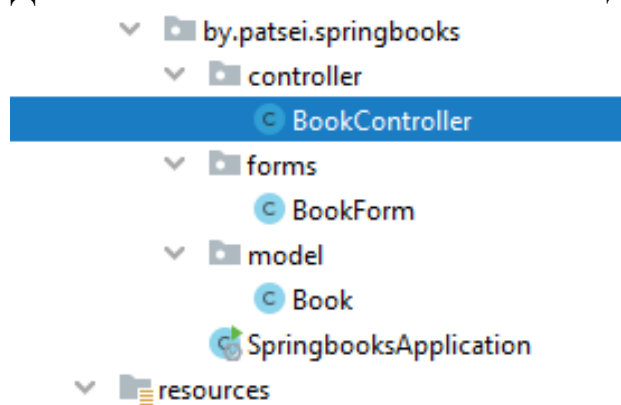
По предыдущему примеру выберите свой тип объектов из таблицы. Варианты «объектов»:

3.	Персона
4.	Дисциплина
5.	Альбом (музыкальный)
6.	Транспорт
7.	Игра
8.	Приложение
9.	Задача
10.	Кинофильм
11.	It -компания

Теперь нужен класс, который будет обрабатывать запросы, т.е. контроллер.

Класс, помеченный как `@Controller` автоматически регистрируется в MVC роутере, а используя аннотации `@(Get|Post|Put|Patch)Mapping` можно регистрировать разные пути. Для REST в Spring есть отдельный тип контроллера который называется `@RestController`, код которого не сильно отличается от обычного контроллера.

Добавим новый пакет *controller* и создадим в нем класс.



BookController является классом **Controller**, который обрабатывает запрос пользователя и управляет потоком приложения.

Аннотация `@Value` - это самый простой способ для “впрыскивания” значений из конфигурации Spring Boot в код. При этом также можно задать значение по-умолчанию.

```
import by.patsei.springbooks.model.Book;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
```

```

import org.springframework.web.servlet.ModelAndView;

import java.util.ArrayList;
import java.util.List;

@Controller
public class BookController {
    private static List<Book> books = new ArrayList<Book>();

    static {
        books.add(new Book("Full Stack Development with JHipster", "Deepu K Sasidharan, Sendil Kumar N"));
        books.add(new Book("Pro Spring Security", "Carlo Scarioni, Massimo Nardone"));
    }

    //
    // Вводится (inject) из application.properties.
    @Value("${welcome.message}")
    private String message;

    @Value("${error.message}")
    private String errorMessage;

    @RequestMapping(value = {"/", "/index"}, method = RequestMethod.GET)
    public ModelAndView index(Model model) {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("index");
        model.addAttribute("message", message);

        return modelAndView;
    }
}

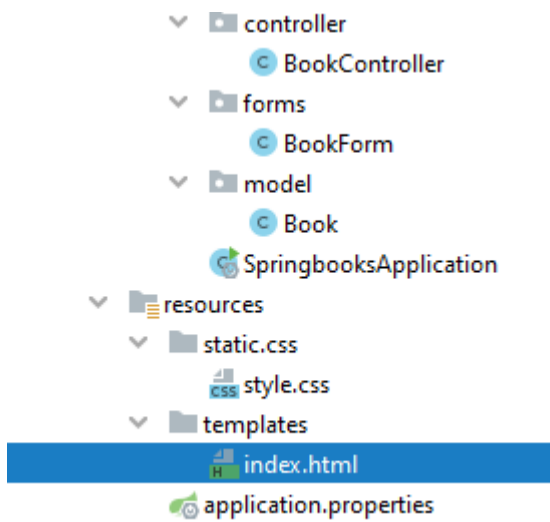
```

У **Spring MVC** есть *DispatcherServlet*. Это главный контроллер, все входящие запросы проходят через него и он уже дальше передает их конкретному контроллеру. Аннотация **@Controller** как раз и сообщает *Spring MVC*, что данный класс является контроллером, диспетчер будет проверять аннотации **@RequestMapping** чтобы вызвать подходящий метод. Аннотация **@RequestMapping** позволяет задать адреса методам контроллера, по которым они будут доступны в клиенте (браузер). Ее можно применять также и к классу контроллера, чтобы задать корневой адрес для всех методов.

Для метода `index()` параметр **value** установлен `{"/", "/index"}`, поэтому он будет вызван сразу, когда в браузере будет набрана комбинация `http://localhost:8080/` или `http://localhost:8080/index`. Параметр *method* указывает какой тип запроса поддерживается (GET, POST, PUT и т.д.). Поскольку тут мы только получаем данные то используется **GET**.

Вместо аннотации **@RequestMapping** с указанием метода, можно использовать аннотации **@GetMapping**, **@PostMapping** и т.д. **@GetMapping** эквивалентно **@RequestMapping(method = RequestMethod.GET)**. В методе создаем объект **ModelAndView** и устанавливаем имя представления, которое нужно вернуть.

Расположите в папке **src/main/resources/templates**



index.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8" />
  <title>Welcome</title>
  <link rel="stylesheet" type="text/css" th:href="@{/css/style.css}"/>
</head>
<body>

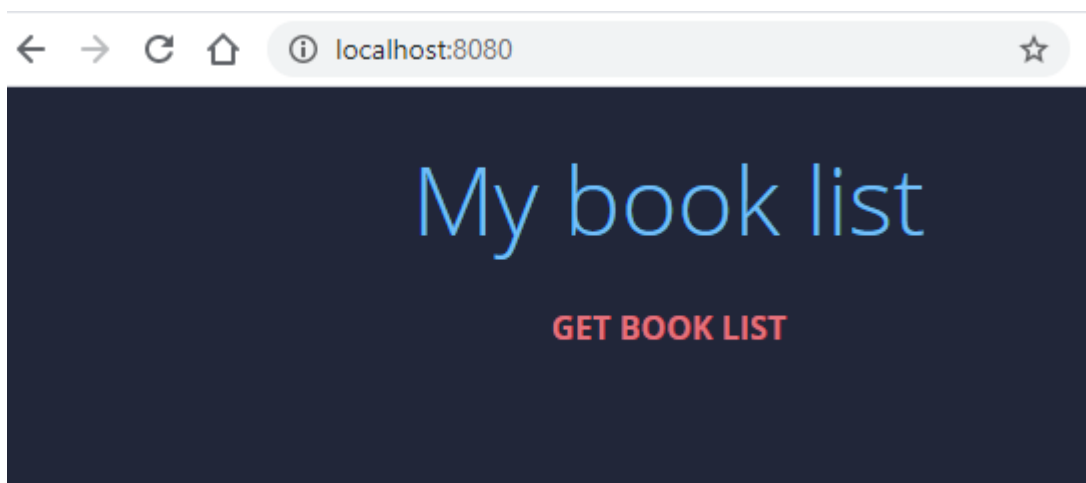
<h1 th:utext="${message}">...!..</h1>
<h2>
<a th:href="@{/allbooks}">Get book list</a>
</h2>

<script src="webjars/jquery/3.3.1/jquery.min.js"></script>
<script src="webjars/bootstrap/4.3.1/js/bootstrap.min.js"></script>

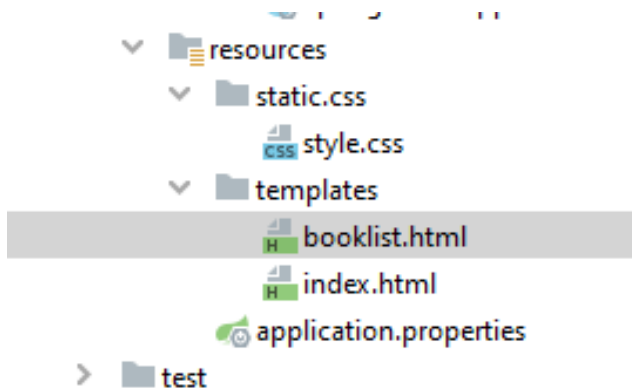
</body>

</html>
```

Запустите приложение. После компиляции проекта — можно сразу идти на <http://localhost:8080> и увидеть созданную страницу. Должно получиться следующее



Создайте **booklist.html**



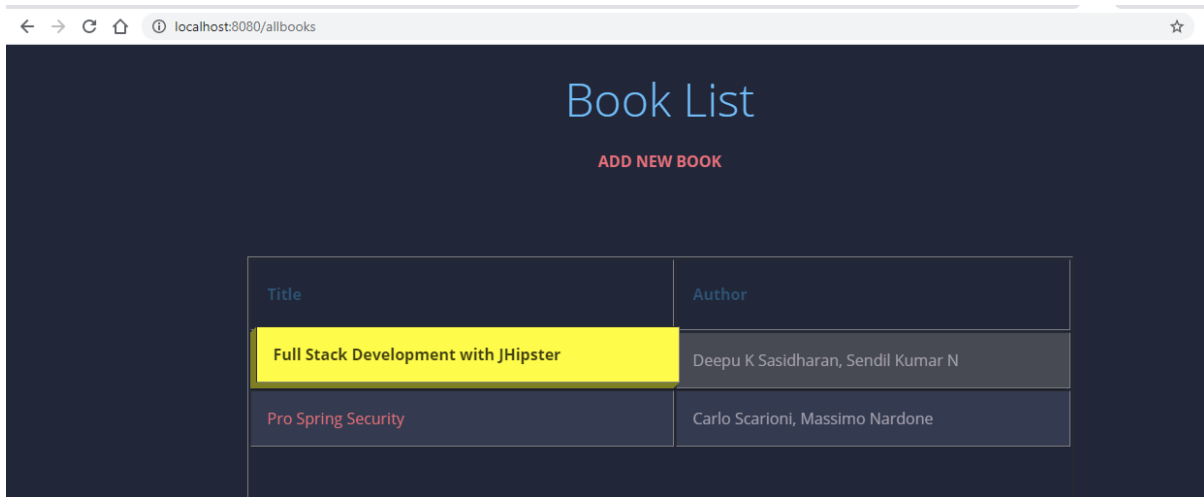
Со следующим содержимым

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8" />
    <title>Book List</title>
    <link rel="stylesheet" type="text/css" th:href="@{/css/style.css}"/>
</head>
<body>
<h1>Book List</h1>
<h2> <a href="addbook">Add new book</a> </h2>
<br/><br/>
    <div class="container">
        <table class="container" border="1">
            <tr>
                <th><h1>Title</h1></th>
                <th><h1>Author</h1></th>
            </tr>
            <tr th:each="book : ${books}">
                <td th:utext="${book.title}">...</td>
                <td th:utext="${book.author}">...</td>
            </tr>
        </table>
    </div>
</body>
</html>
```

Добавьте новый метод в контроллер

```
@RequestMapping(value = {"/allbooks"}, method = RequestMethod.GET)
public ModelAndView personList(Model model) {
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("booklist");
    model.addAttribute("books", books);
    return modelAndView;
}
```

Проверьте выполнение



Создайте **addbook.html**

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8" />
    <title>Add Person</title>
    <link rel="stylesheet" type="text/css" th:href="@{/static/css/style.css}"/>
</head>
<body>
<div class="container">

<h1>New book:</h1>
<form th:action="@{/addbook}"
      th:object="${bookform}" method="POST">
    <p>Title:
      <input type="text" th:field="*{title}" />
    </p>
    <p> Author:
      <input type="text" th:field="*{author}" />
    </p>
    <input class="button-main-page" type="submit" value="Add" />
</form>
</div>
<div th:if="${errorMessage}" th:utext="${errorMessage}"
      style="color:red;font-style:italic;">
</div>
</body>
```

Допишите методы в контроллер

```
@RequestMapping(value = {"/addbook"}, method = RequestMethod.GET)
public ModelAndView showAddPersonPage(Model model) {
    ModelAndView modelAndView = new ModelAndView("addbook");
    BookForm bookForm = new BookForm();
    model.addAttribute("bookform", bookForm);

    return modelAndView;
}
// @PostMapping("/addbook")
// @GetMapping("/")
@RequestMapping(value = {"/addbook"}, method = RequestMethod.POST)
public ModelAndView savePerson(Model model, //
```

```

        @ModelAttribute("bookform") BookForm bookForm) {
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("booklist");
    String title = bookForm.getTitle();
    String author = bookForm.getAuthor();

    if (title != null && title.length() > 0 //
        && author != null && author.length() > 0) {
        Book newBook = new Book(title, author);
        books.add(newBook);
        model.addAttribute("books", books);
        return modelAndView;
    }
    model.addAttribute("errorMessage", errorMessage);
    modelAndView.setViewName("addbook");
    return modelAndView;
}

```

The screenshot shows a web browser window with the address bar displaying 'localhost:8080/addbook'. The main content area has a dark blue background with the title 'New book:' in a large, light blue font. Below the title, there are two input fields: 'Title:' with the text 'Spring in Action' and 'Author:' with the text 'Craig Walls'. At the bottom left of the form is an 'Add' button.

Или

← → ↻ 🏠 ⓘ localhost:8080/addbook

New book:

Title:

Author:

All fields are required

Проверьте что объект добавляется

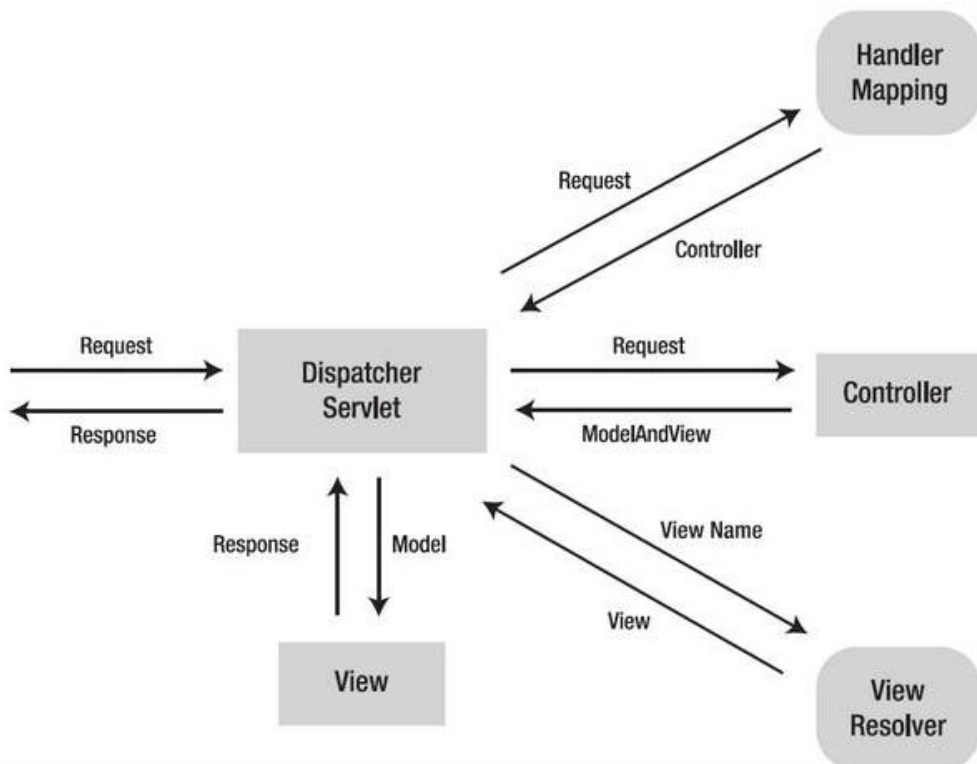
→ ↻ 🏠 ⓘ localhost:8080/addbook ☆ 🌐

Book List

ADD NEW BOOK

Title	Author
Full Stack Development with JHipster	Deepu K Sasidharan, Sendil Kumar N
Pro Spring Security	Carlo Scarioni, Massimo Nardone
Spring/Web	Udemy
Spring in Action	Craig Walls

Мы создали *Spring MVC* приложение, которое работает следующим образом:



Когда мы пишем в строке браузера запрос, его принимает **DispatcherServlet**, далее он находит для обработки этого запроса подходящий контроллер с помощью **HandlerMapping** (это такой интерфейс для выбора контроллера, проверяет в каком из имеющихся контроллеров есть метод, принимающий такой адрес), вызывается подходящий метод и **Controller** возвращает информацию о представлении, затем диспетчер находит нужное представление по имени при помощи **ViewResolver**'а, после чего на это представление передаются данные модели и на выход мы получаем нашу страничку.

Самостоятельно добавьте возможности удаления, редактирования объектов в таблице.

12. Изменение конфигурации

Вернемся к классу `SpringbooksApplication`:

```
@SpringBootApplication
public class SpringbooksApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbooksApplication.class, args);
    }

}
```

Здесь `@SpringBootApplication` - составная аннотация, которая объединяет три другие аннотации Spring:

`@SpringBootApplication` - обозначает класс как класс конфигурации.

Эта аннотация фактически является специализированной формой аннотации @Configuration.

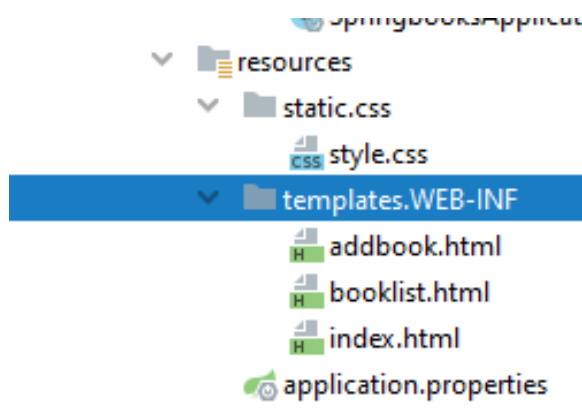
@ EnableAutoConfiguration - включает автоматическую настройку Spring Boot. Эта аннотация говорит Spring Boot автоматически настраивать любые компоненты, которые, по ее мнению, вам понадобятся.

@ ComponentScan - включает сканирование компонентов. Это позволяет объявлять другие классы с аннотациями, такими как @Component, @Controller, @Service и другие, чтобы Spring автоматически обнаруживал их и регистрировал как компоненты в контексте приложения Spring.

Когда Spring Boot обнаруживает зависимость Thymeleaf в POM-файле Maven, он автоматически настраивает механизм шаблонов Thymeleaf.

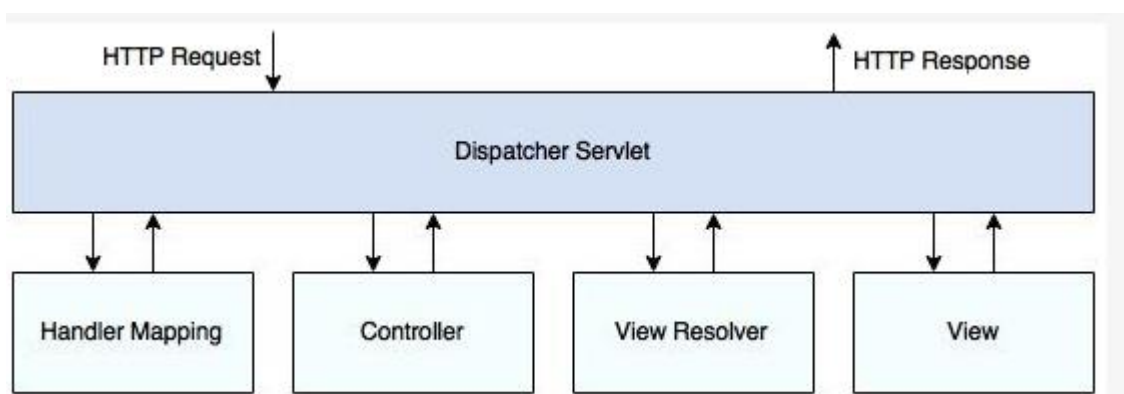
Каталог шаблонов по умолчанию - это src / main / resources / templates.

Сделаем следующее. Создадим папку WEB-INF и поместим туда страницы:

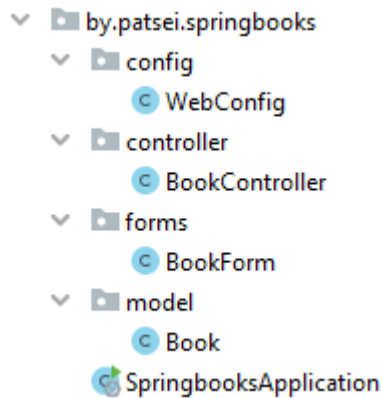


Запустите приложение. На экране должна появиться страница с выводом информации об ошибке. Это происходит потому что ViewResolver не может найти страницу.

Spring Boot автоматически конфигурирует для вас ViewResolver, На рисунке ниже - изображение потока (Flow) приложения Spring в случае когда вы используете ViewResolver (их кстати может быть несколько).



Для конфигурирования **ViewResolver** создайте пакет config, а в нем класс,



например, **WebConfig** аннотированный `@Configuration` со следующим содержанием:

```
@Configuration
public class WebConfig implements WebMvcConfigurer{
    @Bean
    public ClassLoaderTemplateResolver templateResolver() {

        var templateResolver = new ClassLoaderTemplateResolver();

        templateResolver.setPrefix("templates/WEB-INF/");
        templateResolver.setCacheable(false);
        templateResolver.setSuffix(".html");
        templateResolver.setTemplateMode("HTML5");
        templateResolver.setCharacterEncoding("UTF-8");
        return templateResolver;
    }

    @Bean
    public SpringTemplateEngine templateEngine() {
        var templateEngine = new SpringTemplateEngine();
        templateEngine.setTemplateResolver(templateResolver());
        return templateEngine;
    }

    @Bean
    public ViewResolver viewResolver() {
        var viewResolver = new ThymeleafViewResolver();
        viewResolver.setTemplateEngine(templateEngine());
        viewResolver.setCharacterEncoding("UTF-8");
        return viewResolver;
    }

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("index");
    }
}
```

Что мы тут сделали.

`@Configuration` сообщает Spring что данный класс является конфигурационным, содержит определения и зависимости bean-компонентов. Бины (bean) — это объекты, которые управляются Spring'ом. Для определения бина используется аннотация `@Bean`.

Класс **WebConfig** реализует интерфейс **WebMvcConfigurer**, у которого есть целая куча методов, и наставляет все по своему вкусу.

@ComponentScan сообщает Spring где искать компоненты, которыми он должен управлять, т.е. классы, помеченные аннотацией @Component или ее производными, такими как @Controller, @Repository, @Service. Эти аннотации автоматически определяют бин класса.

Первый метод класса **WebConfig** определяет бин преобразователь шаблона:

```
@Bean
public ClassLoaderTemplateResolver templateResolver() {
    var templateResolver = new ClassLoaderTemplateResolver();
    ...
}
```

Средство распознавания шаблонов преобразует шаблоны в объекты TemplateResolution, которые содержат дополнительную информацию, такую как режим шаблона, кэширование, префикс и суффикс шаблонов. ClassLoaderTemplateResolver используется для загрузки шаблонов, расположенных на пути к классам.

Затем устанавливаем каталог шаблонов на:

```
templateResolver.setPrefix("templates/WEB-INF/");
```

Шаблонный движок будет обслуживать контент HTML5:

```
templateResolver.setTemplateMode("HTML5");
```

Определяем остальные свойства.

Потом определяем, что создан шаблонизатор Thymeleaf с интеграцией Spring:

```
@Bean
public SpringTemplateEngine templateEngine() {
    var templateEngine = new SpringTemplateEngine();
    templateEngine.setTemplateResolver(templateResolver());
    return templateEngine;
}
```

Далее настраиваем bean-компонент, который создает ThymeleafViewResolver. Средство разрешения представления отвечает за получение объектов View для конкретной операции и локали. Объекты представления затем визуализируются в файл HTML.

ViewResolver, это интерфейс, необходимый для нахождения представления по имени:

```
@Bean
public ViewResolver viewResolver() {
    var viewResolver = new ThymeleafViewResolver();
    viewResolver.setTemplateEngine(templateEngine());
    viewResolver.setCharacterEncoding("UTF-8");
    return viewResolver;
}
```

Мы определяем автоматический контроллер с помощью метода addViewController ()

```
@Override
public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("index");
}
```

Метод `addViewControllers ()` получает `ViewControllerRegistry`, который можно использовать для регистрации одного или нескольких контроллеров представления. Вызываем `addViewController ()`, передавая `"/"`, то есть путь, по которому контроллер представления будет обрабатывать запросы GET. Этот метод возвращает объект `ViewControllerRegistration`, в котором вызываем `setViewName ()`, чтобы указать начальное представление, на которое должен быть перенаправлен запрос на `«/»`.

Допишите в `application.properties`

```
spring.thymeleaf.prefix=classpath:/templates/WEB-INF/
```

13.Spring Boot SLF4J логгирование

SLF4J (Simple Logging Facade for Java) — библиотека для протоколирования. По умолчанию **SLF4j** уже включен в стартовый пакет Spring Boot.

Настройка логгирования может быть выполнена через **application.properties**. Что бы включить логгирование, изменим `application.properties` файл в корне папки `resources`:

logging.level — определяет уровень логгирования.

```
logging.level.org.springframework.web=ERROR
logging.level.ru.leodev=DEBUG
```

logging.file — определяет имя файла для логгирования, логи будут писаться как в консоль так и в файл одновременно.

```
#создаст файл app.log в папке temp
logging.file.name=${java.io.tmpdir}/app.log

#создаст файл app.log в папке logs Tomcat сервера
#logging.file=${catalina.home}/logs/app.log

#создаст файл app.log по указанному пути
#logging.file=/Users/leo/app.log
```

logging.pattern — определяет собственные правила(шаблон) ведения журнала

```
# паттерн логов для консоли
logging.pattern.console= "%d{yyyy-MM-dd HH:mm:ss} - %msg%n"

# паттерн логов для записи в файл
logging.pattern.file= "%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n"
```

Полный текст файла **application.properties**:

```
spring.thymeleaf.cache=false
spring.thymeleaf.prefix=classpath:/templates/WEB-INF/
```

```
welcome.message = My book list
error.message = All fields are required
```

```
logging.level.org.springframework.web=ERROR
logging.level.ru.leodev=DEBUG
```

```
#создаст файл app.log в папке Logs Tomcat сервера
logging.file.name=${catalina.home}/logs/appSpring.log
```

```
#создаст файл app.log по указанному пути
#logging.file=/Users/Leo/app.log
```

```
# паттерн логов для консоли
logging.pattern.console= "%d{yyyy-MM-dd HH:mm:ss} - %msg%n"
```

```
# паттерн логов для записи в файл
logging.pattern.file= "%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n"
```

Есть альтернативные способы настройки. Например, то же самое можно было бы определить в формате - application.yml.

Можно еще создать стандартный файл logback.xml в корневой папке resources или корне classpath. Это переопределит шаблон логгера Spring Boot.

Аннотируем класс:

```
@Slf4j
@Controller
public class BookController {
    private static List<Book> books = new ArrayList<Book>();
    ...
}
```

@Slf4j, представляет собой аннотацию, предоставленную Lombok, которая во время выполнения автоматически генерирует SLF4J (Simple Logging Facade для Java, [https:// www.slf4j.org/](https://www.slf4j.org/)) Регистратор в классе. Эта аннотация имеет тот же эффект, как если бы вы явно добавили следующие строки в классе:

```
private static final org.slf4j.Logger log =
    org.slf4j.LoggerFactory.getLogger(MainController.class);
```

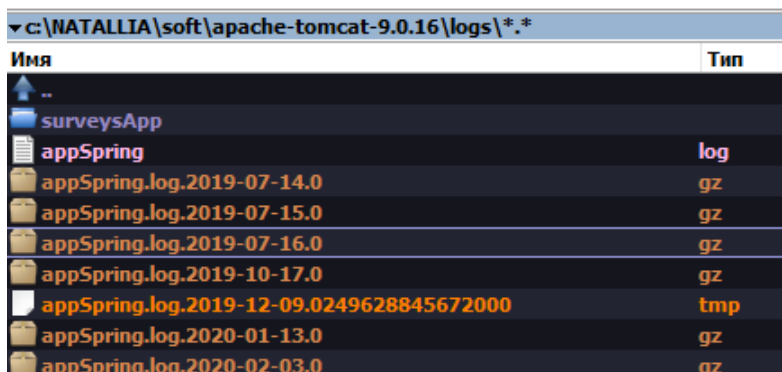
Но аннотации достаточно и можем добавить log, например к классу контроллера в методы:

```
@RequestMapping(value = {"/", "/index"}, method = RequestMethod.GET)
public ModelAndView index(Model model) {
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("index");
    model.addAttribute("message", message);
    log.info("/index was called");
    return modelAndView;
}
```

Запустите приложение и выполните несколько переходов по страницам. На консоли вы увидите:

```
""2020-08-31 23:51:07 - HHH000490: Using JtaPlatform implementation: [org.hibernate
""2020-08-31 23:51:07 - Initialized JPA EntityManagerFactory for persistence unit '
""2020-08-31 23:51:24 - Initializing Spring DispatcherServlet 'dispatcherServlet'
""2020-08-31 23:51:24 - /index was called
""2020-08-31 23:51:27 - /booklist was called
""2020-08-31 23:51:28 - /addbook GET was called
""2020-08-31 23:51:33 - /addbook GET was called
""2020-08-31 23:51:35 - /addbook POST was called
""2020-08-31 23:51:41 - /index was called
""
```

Также согласно настройкам в Tomcat должен появиться файл **appSpring**:



со следующим содержимым:

```
Lister - [C:\NATALIA\soft\apache-tomcat-9.0.16\logs\appSpring.log]
Файл  Правка  Вид  Кодировка  Справка
""2020-09-01 00:00:47 [restartedMain] INFO b.p.s.SpringbooksApplication - Starting SpringbooksApplication on DESKTOP-
""2020-09-01 00:00:47 [restartedMain] INFO b.p.s.SpringbooksApplication - No active profile set, falling back to def
""2020-09-01 00:00:47 [restartedMain] INFO o.s.b.d.e.DevToolsPropertyDefaultsPostProcessor - Devtools property defau
""2020-09-01 00:00:47 [restartedMain] INFO o.s.b.d.e.DevToolsPropertyDefaultsPostProcessor - For additional web rela
""2020-09-01 00:00:49 [restartedMain] INFO o.s.d.r.c.RepositoryConfigurationDelegate - Bootstrapping Spring Data JPA
""2020-09-01 00:00:49 [restartedMain] INFO o.s.d.r.c.RepositoryConfigurationDelegate - Finished Spring Data reposito
""2020-09-01 00:00:50 [restartedMain] INFO o.s.b.w.e.tomcat.TomcatWebServer - Tomcat initialized with port(s): 8080
""2020-09-01 00:00:50 [restartedMain] INFO o.a.catalina.core.StandardService - Starting service [Tomcat]
""2020-09-01 00:00:50 [restartedMain] INFO o.a.catalina.core.StandardEngine - Starting Servlet engine: [Apache Tomca
""2020-09-01 00:00:50 [restartedMain] INFO o.a.c.c.C.[Tomcat].[localhost].[/] - Initializing Spring embedded WebAppl
""2020-09-01 00:00:50 [restartedMain] INFO o.s.b.w.s.c.ServletWebServerApplicationContext - Root WebApplicationConte
""2020-09-01 00:00:50 [restartedMain] INFO com.zaxxer.hikari.HikariDataSource - HikariPool-1 - Starting...
""2020-09-01 00:00:50 [restartedMain] INFO com.zaxxer.hikari.HikariDataSource - HikariPool-1 - Start completed.
""2020-09-01 00:00:50 [restartedMain] INFO o.s.b.a.h.H2ConsoleAutoConfiguration - H2 console available at '/h2-conso
""2020-09-01 00:00:51 [restartedMain] INFO o.s.s.c.ThreadPoolTaskExecutor - Initializing ExecutorService 'applicatio
""2020-09-01 00:00:51 [task-1] INFO o.h.jpa.internal.util.LogHelper - HHH000204: Processing PersistenceUnitInfo [nam
""2020-09-01 00:00:51 [restartedMain] WARN o.t.template.TemplateMode - [THYMELEAF][restartedMain] Template Mode
""2020-09-01 00:00:51 [task-1] INFO org.hibernate.Version - HHH000412: Hibernate ORM core version 5.4.20.Final
""2020-09-01 00:00:51 [restartedMain] WARN o.s.b.a.o.j.JpaBaseConfiguration$JpaWebConfiguration - spring.jpa.open-in
""2020-09-01 00:00:51 [task-1] INFO o.h.annotations.common.Version - HCANN000001: Hibernate Commons Annotations {5.1
""2020-09-01 00:00:51 [restartedMain] INFO o.s.b.a.w.s.WelcomePageHandlerMapping - Adding welcome page template: ind
""2020-09-01 00:00:51 [task-1] INFO org.hibernate.dialect.Dialect - HHH000400: Using dialect: org.hibernate.dialect.
""2020-09-01 00:00:52 [restartedMain] INFO o.s.b.d.a.OptionalLiveReloadServer - LiveReload server is running on port
""2020-09-01 00:00:52 [restartedMain] INFO o.s.b.w.e.tomcat.TomcatWebServer - Tomcat started on port(s): 8080 (http
""2020-09-01 00:00:52 [restartedMain] INFO o.s.d.r.c.DeferredRepositoryInitializationListener - Triggering deferred
""2020-09-01 00:00:52 [restartedMain] INFO o.s.d.r.c.DeferredRepositoryInitializationListener - Spring Data reposito
""2020-09-01 00:00:52 [task-1] INFO o.h.e.t.j.p.i.JtaPlatformInitiator - HHH000490: Using JtaPlatform implementation
""2020-09-01 00:00:52 [task-1] INFO o.s.o.j.LocalContainerEntityManagerFactoryBean - Initialized JPA EntityManagerFa
""2020-09-01 00:00:52 [restartedMain] INFO b.p.s.SpringbooksApplication - Started SpringbooksApplication in 6.107 se
""2020-09-01 00:01:00 [http-nio-8080-exec-1] INFO o.a.c.c.C.[Tomcat].[localhost].[/] - Initializing Spring Dispatche
""2020-09-01 00:01:02 [http-nio-8080-exec-6] INFO b.p.s.controller.BookController - /index was called
""2020-09-01 00:01:03 [http-nio-8080-exec-8] INFO b.p.s.controller.BookController - /booklist was called
""2020-09-01 00:01:04 [http-nio-8080-exec-10] INFO b.p.s.controller.BookController - /addbook GET was called
""2020-09-01 00:01:06 [http-nio-8080-exec-2] INFO b.p.s.controller.BookController - /addbook POST was called
""2020-09-01 00:01:11 [http-nio-8080-exec-5] INFO b.p.s.controller.BookController - /addbook GET was called
```

14.Адресация в Контроллере

Посмотрим еще раз на класс Контроллера и попробуем использовать другие аннотации.

Спецификация класса `@RequestMapping` может уточняется с помощью аннотации: `@GetMapping`. `@GetMapping` в паре с классом уровня `@RequestMapping` указывает, что при получении запроса HTTP GET этот метод будет вызван для обработки запроса.

`@GetMapping` - это относительно новая аннотация, появившаяся в Spring 4.3. До Spring 4.3 могли использовать аннотацию `@RequestMapping` уровня метода:

```
@RequestMapping (method=RequestMethod.GET)
```

Очевидно, что `@GetMapping` более лаконичен и специфичен для метода HTTP, на который он нацелен. Однако, `@GetMapping` - всего лишь одна из семейства аннотаций отображения запросов. В Таблице 1 перечислены все аннотации отображения запросов, доступные в Spring MVC.

Таблица

Аннотации	Описание
<code>@RequestMapping</code>	Обработка запросов общего назначения
<code>@GetMapping</code>	Обработка GET запросов
<code>@PostMapping</code>	Обработка POST запросов
<code>@PutMapping</code>	Обработка PUT запросов
<code>@DeleteMapping</code>	Обработка DELETE запросов
<code>@PatchMapping</code>	Обработка PATCH запросов

Новые аннотации сопоставления запросов имеют все те же атрибуты, что и `@RequestMapping`, так что вы можете использовать их везде, где использовали `@RequestMapping`.

Обычно `@RequestMapping` используется на уровне класса. А более конкретные `@GetMapping`, `@PostMapping` и т.д. аннотации используются на каждом из методов-обработчиков.

Перепишем аннотации класса контроллера следующим образом:

```
@Slf4j
@Controller
@RequestMapping
public class MainController {

    @GetMapping(value = {"/", "/index"})
    public ModelAndView index(Model model) {
    ...
    @GetMapping(value = {"/allbooks"})
    public ModelAndView personList(Model model) {
    ...
    @GetMapping(value = {"/addbook"})
    public ModelAndView showAddPersonPage(Model model) {
    ...
    @PostMapping(value = {"/addbook"})
    public ModelAndView savePerson(Model model, //
    ...
```

Запустите приложение. Проверьте все переходы.

Вопросы.

1. Перечислите Spring модули и их назначение.
2. Расскажите о составе Spring Framework.
3. Что такое Spring Boot? В чем его преимущества и для чего он используется?
4. Для чего используется аннотация `@SpringBootApplication`?
5. Объясните принцип IoC (Inversion of Control)? Какие формы используются в Spring для внедрения?
6. В чем суть понятия Inversion of Control (IoC)?
7. В чем различие внедрение зависимостей (Dependency Injection) и поиска зависимостей (Dependency Lookup)?
8. Что такое JavaBean? Какие есть правила описания и использования?
9. Перечислите области видимости bean.
10. Опишите ЖЦ бина.
11. Поясните значения аннотаций: `@Configuration`, `@Bean`, `@Component`, `@Service`, `@Repository`, `@Controller`.
12. Spring Expression Language (SpEL): расскажите об особенностях и области использования.
13. Охарактеризуйте основные Core Container Spring.
14. Как в Spring происходит разрешение зависимостей?
15. Поясните как работает `DispatcherServlet`, `HandlerMapping`, `ViewResolver`? Как происходит обработка запроса?
16. Как используется паттерн «Front Controller» в Spring?
17. Как происходит адресация в контроллере?
18. Расскажите про Spring MVC архитектуру.
19. За что отвечает `WebApplicationContext`?