

Тема 2 . Spring. Spring Framework. Введение

Литература

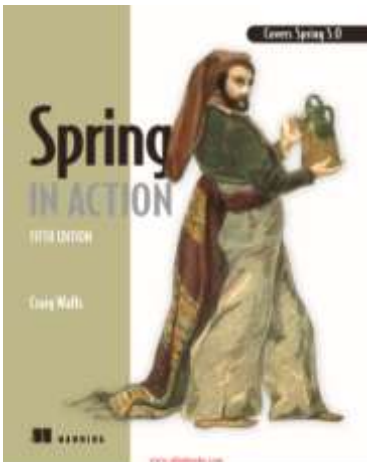
<https://spring.io/>

<https://www.manning.com/books/spring-in-action-fourth-edition>

<https://docs.spring.io/spring/docs/current/spring-framework-reference/index.html>

<https://www.manning.com/books/spring-in-action-fourth-edition>





Learn Spring 5.0

Version 5.1.9.RELEASE

Spring Framework Documentation

What's New, Upgrade Notes, Supported Versions, and other topics, independent of release cadence, are maintained externally on the project's [Github Wiki](#).

Overview	history, design philosophy, feedback, getting started.
Core	IoC Container, Events, Resources, i18n, Validation, Data Binding, Type Conversion, SpEL, AOP.
Testing	Mock Objects, TestContext Framework, Spring MVC Test, WebTestClient.
Data Access	Transactions, DAO Support, JDBC, O/R Mapping, XML, Marshalling.
Web Servlet	Spring MVC, WebSocket, SockJS, STOMP Messaging.
Web Reactive	Spring WebFlux, WebClient, WebSocket.

История Spring

Был разработан в 2003 года Родом Джонсоном.

Spring дополняет Java EE. Модель программирования Spring не охватывает спецификацию платформы Java EE; интегрируется с EE:

Servlet API ([JSR 340](#))

WebSocket API ([JSR 356](#))

Concurrency Utilities ([JSR 236](#))

JSON Binding API ([JSR 367](#))

Bean Validation ([JSR 303](#))

JPA ([JSR 338](#))

JMS ([JSR 914](#))

Spring Framework также поддерживает спецификации

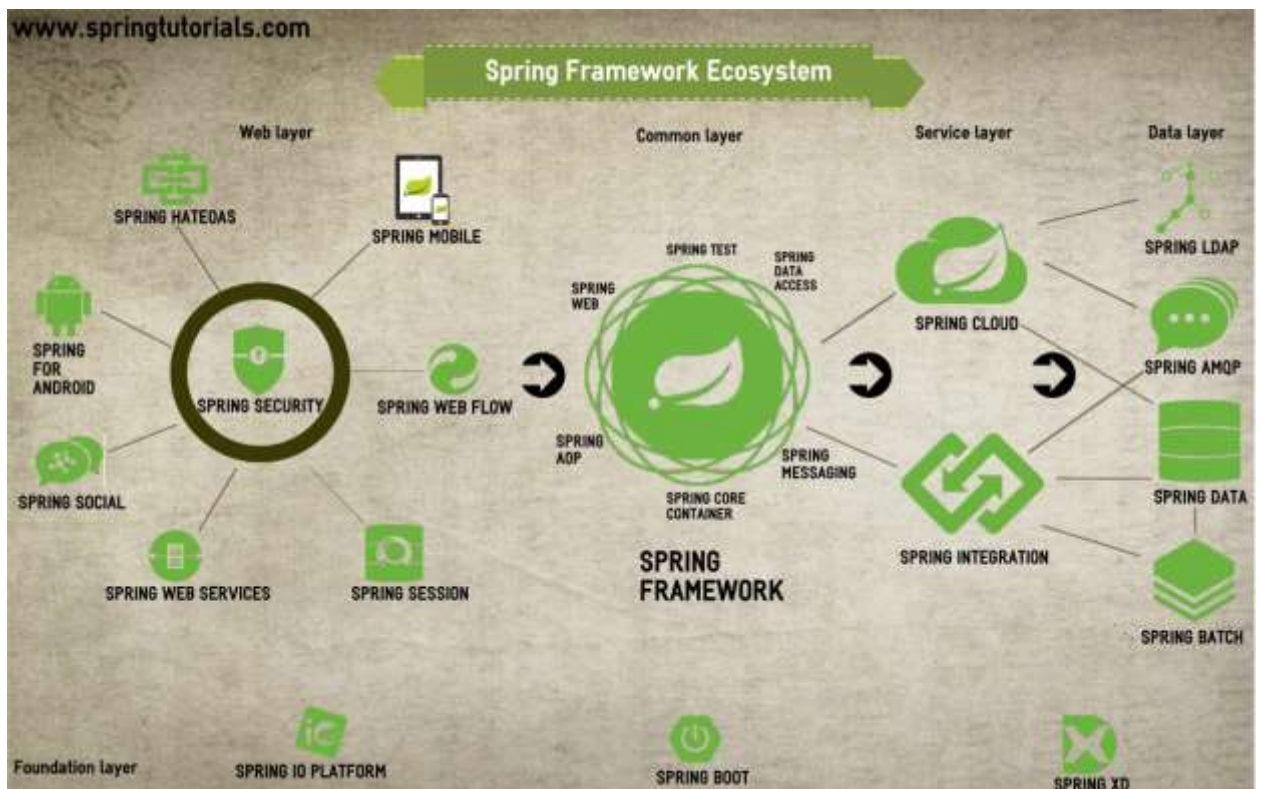
Dependency Injection ([JSR 330](#)) and Common Annotations ([JSR 250](#)) specifications, которые разработчики приложений могут использовать вместо механизмов Spring, предоставляемых Spring Framework.

На веб-сайте Spring в настоящее время перечислены >20 проекта Spring.

Некоторые проекты Spring имеют подпроекты.

Модуль Spring - это полная и независимая функция с минимальной зависимостью от других.

Spring Экосистема



- 1) Веб-уровень, 2) Общий уровень, 3) Уровень обслуживания и 4) Уровень данных. Три проекта внизу страницы - пятая часть. и могут охватывать много слоев.

В настоящее время, под термином "Spring" часто подразумевают целое семейство проектов.

Spring Framework (или Spring Core) Ядро платформы, предоставляет базовые средства для создания приложений — управление компонентами (бинами, **beans**), внедрение зависимостей, MVC фреймворк, транзакции, базовый доступ к БД. В основном это низкоуровневые компоненты и абстракции. По сути, неявно используется всеми другими компонентами.

Spring MVC (часть Spring Framework) Оперировать понятиями контроллеров, маппингов запросов, различными HTTP абстракциями и т.п. Со Spring MVC интегрированы шаблонные движки, типа Thymeleaf, Freemarker, Mustache, плюс есть сторонние интеграции. JSP или JSF писать не нужно.

Spring Data Доступ к данным: реляционные и нереляционные БД, KV хранилища и т.п.

Spring Cloud Много полезного для микросервисной архитектуры — service discovery, трассировка и диагностика, балансировщики запросов, circuit breaker-ы, роутеры и т.п.

Spring Security Авторизация и аутентификация, доступ к данным, методам и т.п. OAuth, LDAP, и куча разных провайдеров.

Spring Integration Обработка данных из разных источников.

Типичное веб приложение скорее всего будет включать набор вроде Spring MVC, Data, Security.

Особняком стоит отметить **Spring Boot**. Boot позволяет быстро создать и сконфигурировать (т.е. настроить зависимости между компонентами) приложение, упаковать его в исполняемый самодостаточный артефакт. Это то связующее звено, которое объединяет вместе набор компонентов в готовое приложение. Не использует XML для конфигурации. Все конфигурируется через аннотации.

Чтобы ускорить процесс управления зависимостями, Spring Boot неявно упаковывает необходимые сторонние зависимости для каждого типа приложения на основе Spring и предоставляет их разработчику посредством так называемых starter-пакетов (spring-boot-starter-web, spring-boot-starter-data-jpa и т.д.)

Starter-пакеты представляют собой набор удобных дескрипторов зависимостей, которые можно включить в свое приложение. Это позволит получить универсальное решение для всех, связанных со Spring технологий, избавляя нас от лишнего поиска примеров кода и загрузки из них требуемых дескрипторов зависимостей.

Например, если вы хотите начать использовать Spring Data JPA для доступа к базе данных, просто включите в свой проект зависимость spring-boot-starter-data-jpa и все будет готово (вам не придется искать совместимые драйверы баз данных и библиотеки Hibernate)

Если вы хотите создать Spring web-приложение, просто добавьте зависимость spring-boot-starter-web, которая подтянет в проект все библиотеки, необходимые для разработки Spring MVC-приложений, таких как spring-webmvc, jackson-json, validation-api и Tomcat

Другими словами, Spring Boot собирает все общие зависимости и определяет их в одном месте.

Следовательно, при использовании Spring Boot, файл pom.xml содержит намного меньше строк, чем при использовании его в Spring-приложениях

Второй превосходной возможностью **Spring Boot** является автоматическая конфигурация приложения. После выбора подходящего **starter**-пакета, **Spring Boot** попытается автоматически настроить Spring-приложение на основе добавленных вами **jar**-зависимостей

Например, если вы добавите **Spring-boot-starter-web**, Spring Boot автоматически сконфигурирует такие зарегистрированные бины, как **DispatcherServlet**, **ResourceHandlers**, **MessageSource**

Если вы используете **spring-boot-starter-jdbc**, **Spring Boot** автоматически регистрирует бины **DataSource**, **EntityManagerFactory**, **TransactionManager** и считывает информацию для подключения к базе данных из файла **application.properties**

Если вы не собираетесь использовать базу данных, и не предоставляете никаких подробных сведений о подключении в ручном режиме, Spring Boot

автоматически настроит базу в памяти, без какой-либо дополнительной конфигурации с вашей стороны (при наличии H2 или HSQL библиотек)

.Автоматическая конфигурация может быть полностью переопределена в любой момент с помощью пользовательских настроек.

Каждое Spring Boot web-приложение включает встроенный web-сервер. Ниже приведен список контейнеров сервлетов, которые поддерживаются "из коробки"

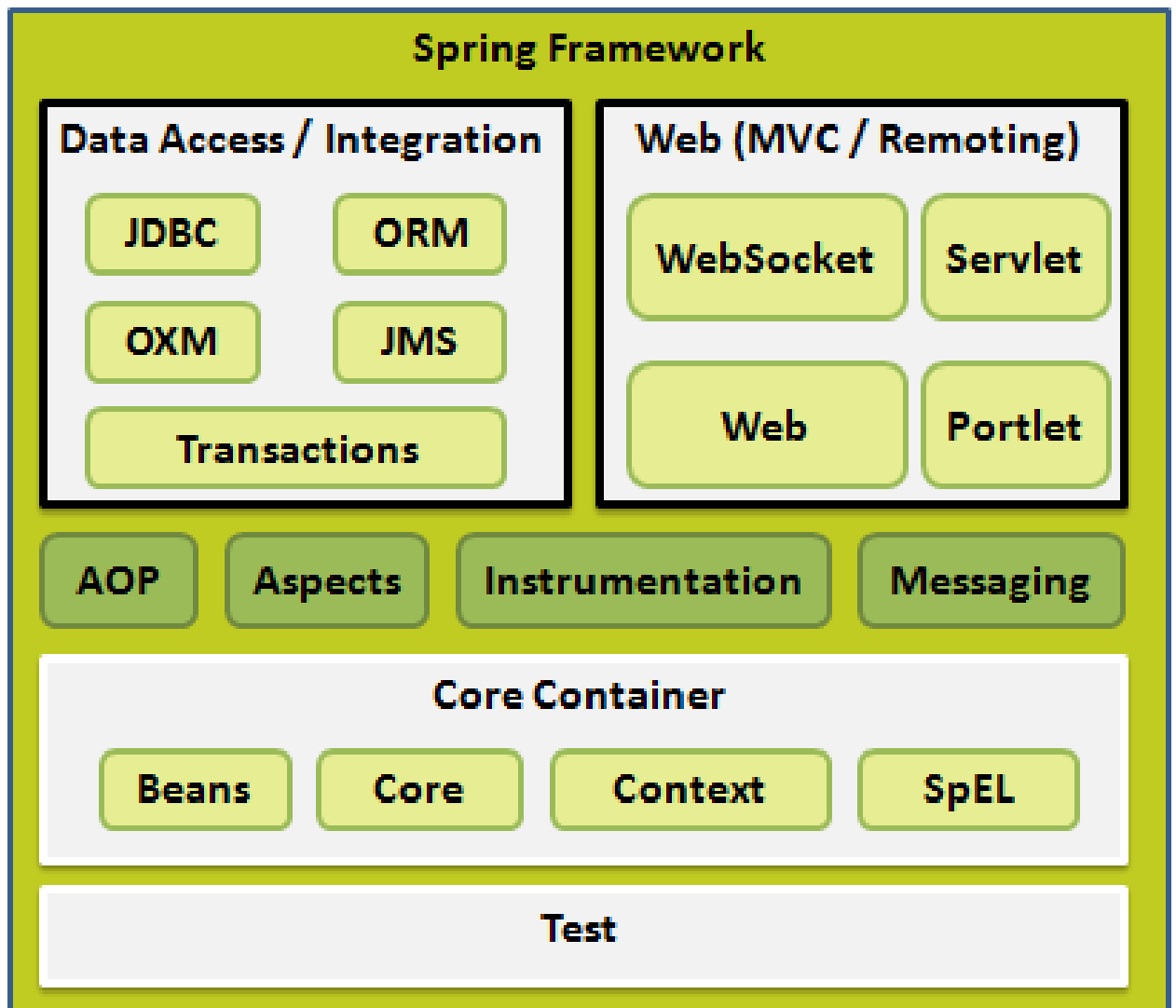
Name	Servlet Version
Tomcat 9.0	4.0
Jetty 9.4	3.1
Undertow 2.0	4.0

Теперь не надо беспокоиться о настройке контейнера сервлетов и развертывании приложения на нем. Приложение может запускаться само, как исполняемый jar-файл с использованием встроенного сервера

Если вам нужно использовать отдельный HTTP-сервер, для этого достаточно исключить зависимости по умолчанию. Spring Boot предоставляет отдельные starter-пакеты для разных HTTP-серверов

Создание автономных web-приложений со встроенными серверами не только удобно для разработки, но и является допустимым решением для приложений корпоративного уровня и становится все более полезно в мире микросервисов.

SpringFramework



Основной контейнер (Core Container) включает в себя Beans, Core, Context и SpEL (expression language).

Beans отвечает за BeanFactory которая является сложной реализацией паттерна Фабрика (GoF) Это— базовые примитивы Spring namespace, включая декларирование бинов и как они должны быть связаны.

Модуль Core обеспечивает ключевые части фреймворка, включая свойства IoC и DI.

Context построен на основе Beans и Core и позволяет получить доступ к любому объекту, который определён в настройках. Ключевым элементом модуля Context является интерфейс ApplicationContext.

Модуль SpEL обеспечивает язык выражений для манипулирования объектами во время исполнения.

AOP реализует аспекто-ориентированное программирование .

Модуль Aspects обеспечивает интеграцию с AspectJ.

Instrumentation отвечает за поддержку class instrumentation и class loader, которые используются в серверных приложениях.

Модуль **Messaging** обеспечивает поддержку STOMP.

Модуль **Test** обеспечивает тестирование с использованием TestNG или JUnit Framework.

Контейнер **Data Access/Integration** состоит из JDBC, ORM, OXM, JMS и модуля Transactions.

JDBC обеспечивает абстрактный слой JDBC.

ORM обеспечивает интеграцию с ORM, например Hibernate, JDO, JPA и т.д.

Модуль **OXM** отвечает за связь Объект/XML – XMLBeans, JAXB и т.д.

Модуль **JMS** (*Java Messaging Service*) отвечает за создание, передачу и получение сообщений.

Transactions поддерживает управление транзакциями для классов, которые реализуют определённые методы.

Web. Этот слой состоит из Web, Web-MVC, Web-Socket, Web-Portlet

Модуль **Web** обеспечивает такие функции, как загрузка файлов и т.д.

Web-MVC содержит реализацию Spring MVC для веб-приложений.

Spring MVC стоит упомянуть отдельно, т.к. мы будем вести речь в основном о веб-приложениях. Он оперирует понятиями контроллеров, маппингов запросов, различными HTTP абстракциями и т.п. Со Spring MVC интегрированы нормальные шаблонные движки, типа Thymeleaf, Freemaker, Mustache, плюс есть сторонние интеграции с кучей других. Так что никаких JSP или JSF писать не нужно.

Web-Socket обеспечивает поддержку связи между клиентом и сервером, используя Web-Socket-ы в веб-приложениях.

Web-Portlet обеспечивает реализацию MVC в среде портлетов.

Spring Initializr. Spring Boot

Для того, чтобы создать простое приложение нужно знать как создать проект Maven с нуля, как настроить плагины, чтобы создать JAR, как установить и запустить локально Tomcat, ну и как работает DispatcherServlet.

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

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

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.

← → ↻ <https://start.spring.io>

 **Spring Initializr**
Bootstrap your application

Project

Language

Spring Boot

Project Metadata

Maven Project **Gradle Project**

Java Kotlin Groovy

2.2.0 M2 2.2.0 (SNAPSHOT) 2.1.5 (SNAPSHOT) 2.1.4 1.5.20

Group
by.patsei

Artifact
SpringProject1

Name
SpringProject1

Description
Demo project for Spring Boot

Description
Demo project for Spring Boot

Package Name
by.patsei.SpringProject1

Packaging
Jar War

Java Version
12 **11** 8

Fewer options

Нажмите Создать проект. Импортируйте проект в IntelliJ Idea.
Второй способ можно это же сделать при создании проекта в IntelliJ Idea.
Выберите SpringInitializr

Java

Java Enterprise

JBoss

J2ME

Clouds

Spring

Java FX

Android

IntelliJ Platform Plugin

Spring Initializr

Maven

Gradle

Groovy

Grails

Application Forge

Static Web

Flash

Project SDK: 11.0.2

New...

Choose Initializr Service URL.

☒ Default: <https://start.spring.io>

☐ Custom:

Make sure your network connection is active before continuing.

Previous

Next

Cancel

Help

New Project

Project Metadata

Group: by.patsei

Artifact: springproject1

Type: Maven Project (Generate a Maven based project archive)

Language: Java

Packaging: War

Java Version: 11

Version: 0.0.1-SNAPSHOT

Name: springproject1

Description: Demo project for Spring Boot

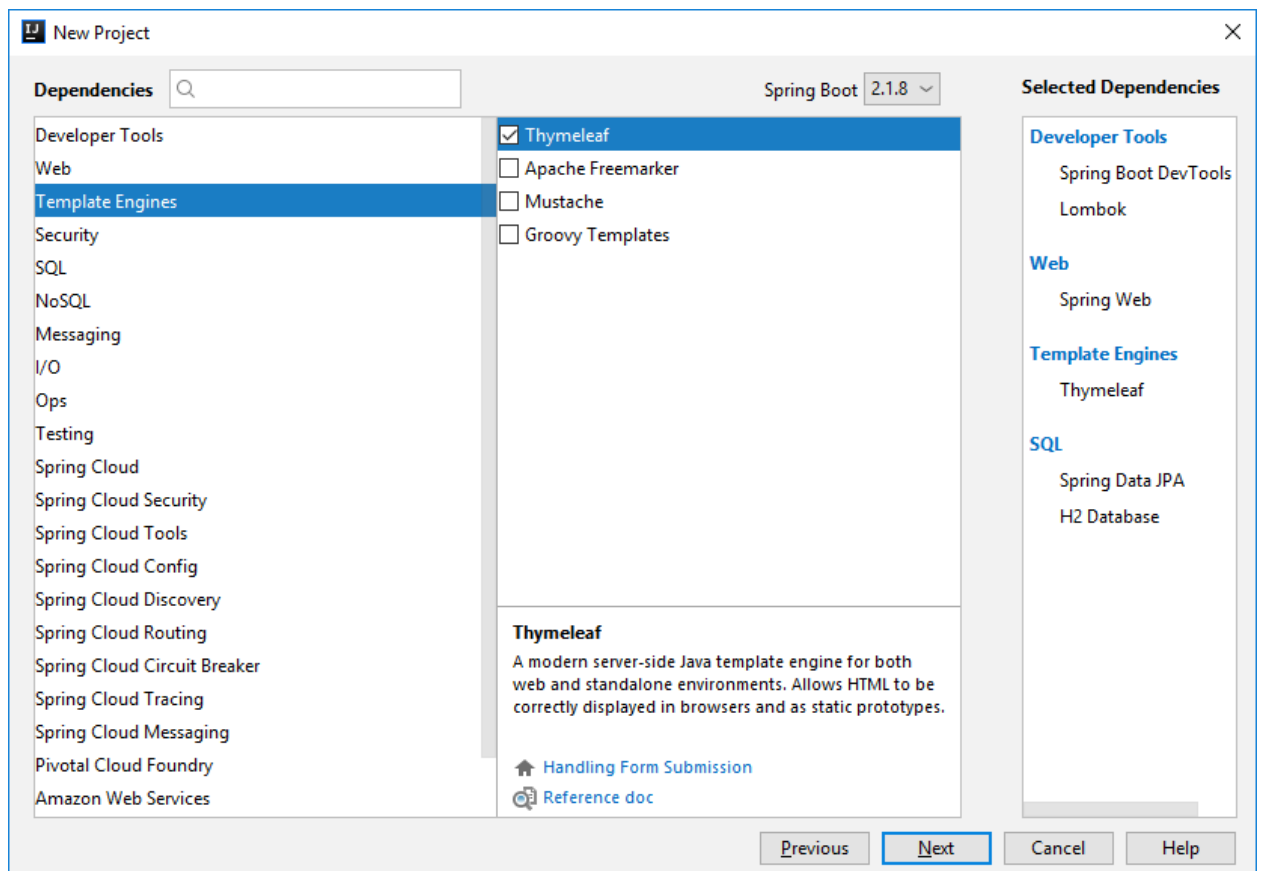
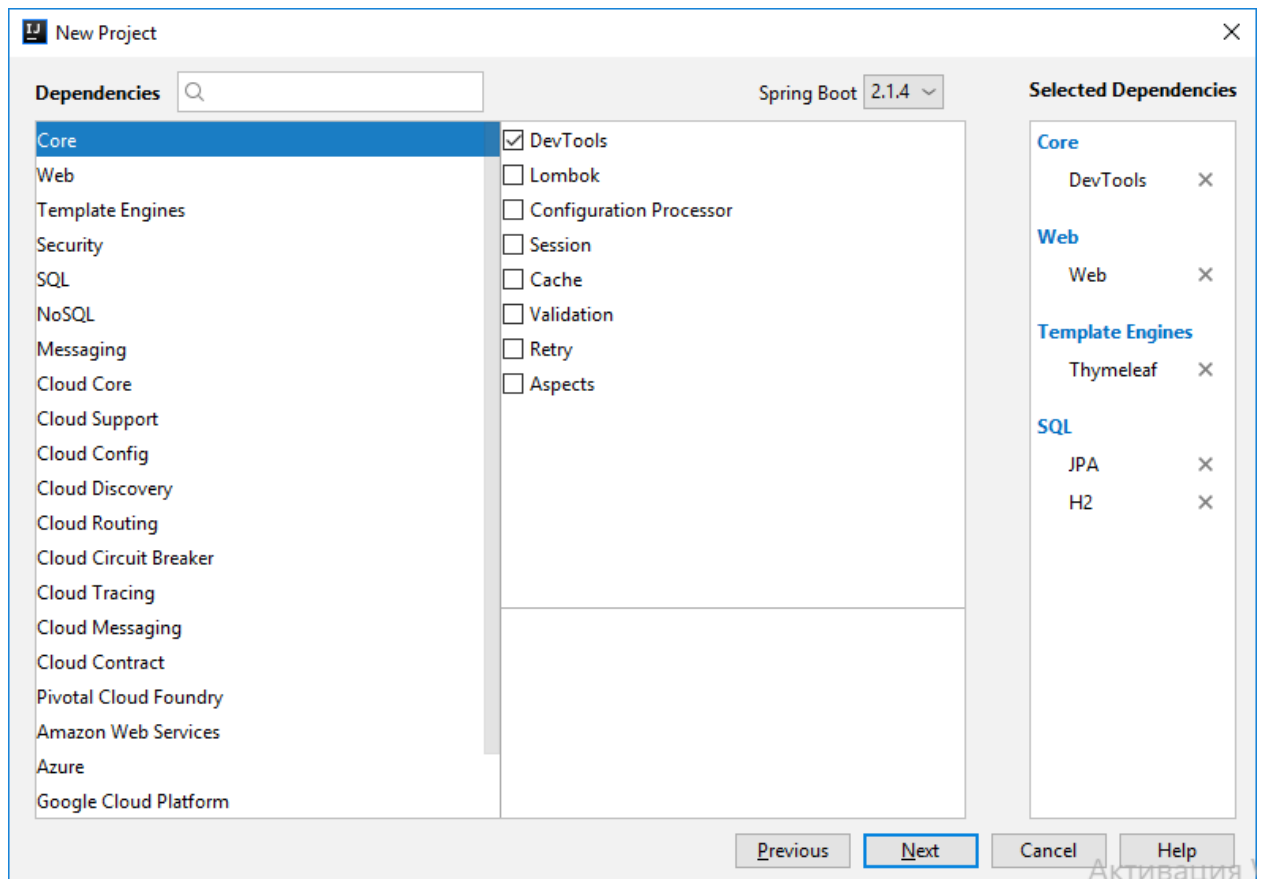
Package: by.patsei.springproject1

Previous

Next

Cancel

Help



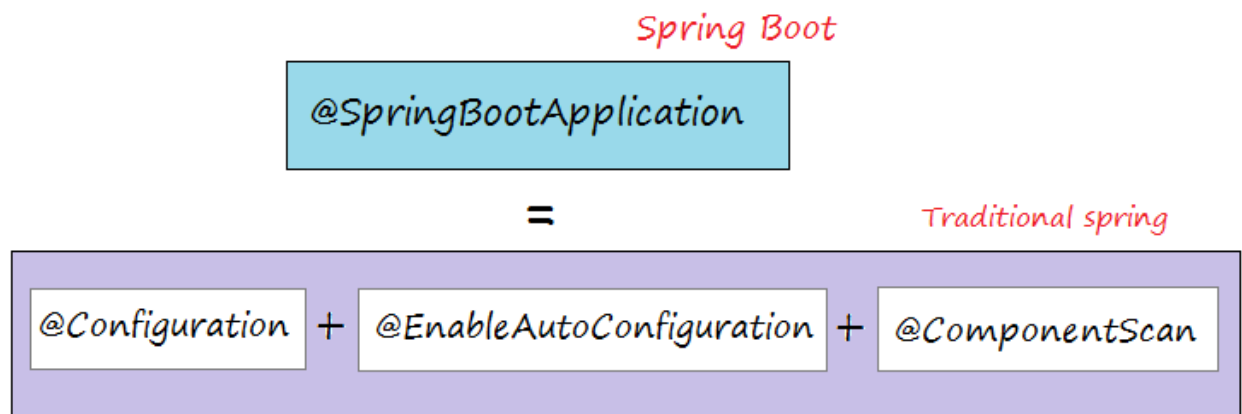
Запустите проект. Изучите структуру проекта.

Ваше приложение начинается выполнением класса
SpringProject1Application.

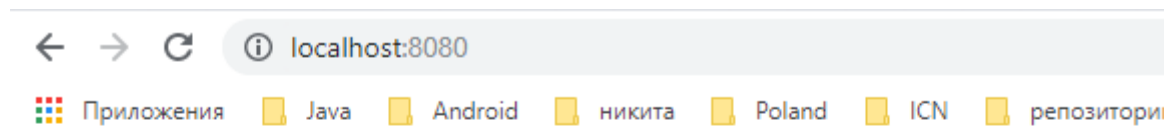
```
@SpringBootApplication
```

```
public class SpringProject1Application {  
    public static void main(String[] args) {  
        SpringApplication.run(SpringProject1Application.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, ...)



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



Whitelabel Error Page

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

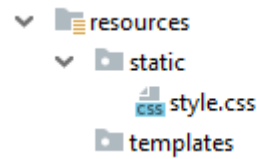
Tue Sep 17 11:10:28 MSK 2019

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

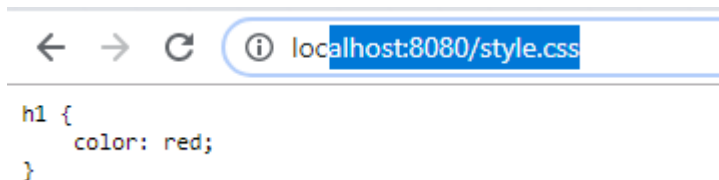
No message available

Для статических ресурсов (Static Resource), например файлов **css, javascript, image**,... существует папка **src/main/resources/static**

или в размещать можно в подпапках.



Запустите проект и обратитесь следующим образом:



Добавьте в папку **static** файл со стилями - **style.css**

2. Создадим контроллер и вернем домашнюю страницу

Класс, помеченный как **@Controller** автоматически регистрируется в MVC роутере, а используя аннотации **@(Get|Post|Put|Patch)Mapping** можно регистрировать разные пути.

```
package by.patsei.springproject1;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.servlet.ModelAndView;
import java.util.HashMap;
import java.util.Map;

@Controller
public class IndexController {

    @GetMapping("/")
    public ModelAndView index() {
        Map<String, String> model = new HashMap<>();
        model.put("name", "ПОИТ 3 курс");
        return new ModelAndView("index", model);
    }
}
```

Метод возвращает **ModelAndView** — дальше Spring знает, что нужно взять **view** **index.html** из папки **resources/templates** (это соглашение по умолчанию) и передать туда модель

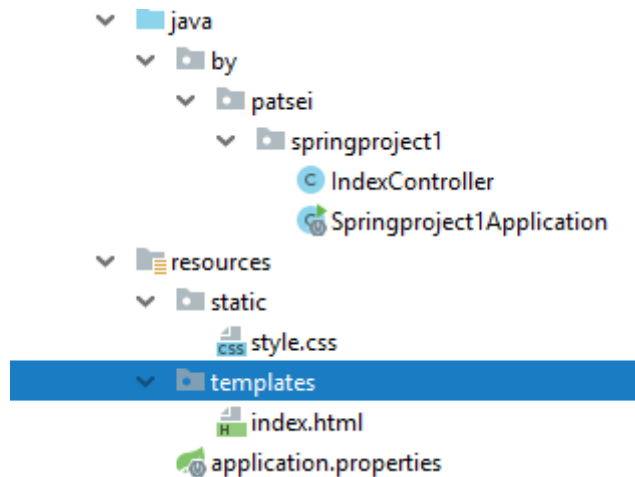
Модель в нашем случае просто словарь, но это может быть и строго-типизированная модель (объект) поэтому можно записать так:

```
public ModelAndView index() {
    // Map<String, String> model = new HashMap<>();
    // model.put("name", "ПОИТ 3 курс");
    // return new ModelAndView("index", model);

    ModelAndView modelAndView = new ModelAndView("index");
    modelAndView.addObject("name", "ПОИТ 3 курс");
    return modelAndView;
}
```

Этот интерфейс позволяет нам передавать всю информацию, требуемую Spring MVC, за один возврат.

3. Создаем страницу



Все данные, которые мы помещаем в model, используются view - в общем, шаблонным представлением для визуализации веб-страницы. Если есть файл шаблона Thymeleaf, параметр, передаваемый через модель, будет доступен из HTML-кода:

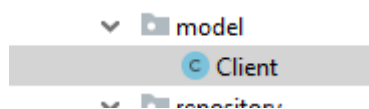
```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8" />
  <title>Welcome</title>
  <link rel="stylesheet" type="text/css" th:href="@{/css/style.css}"/>
</head>
<body>
<h1>
  Welcome to Spring,
  <h1 th:text="${name}"/>
</h1>
</body>
</html>
```

После компиляции проекта можно сразу идти на <http://localhost:8080> и увидеть созданную страницу

Welcome to Spring,

ПОИТ 3 курс

4. Добавим данные



Это Spring Data. Мы будем собирать статистику посещений — каждый раз, когда кто-то заходит на главную страницу, мы будем писать это в базу. Модель выглядит так:

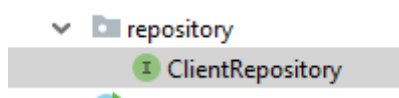
```
import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
@Getter
@Setter
@NoArgsConstructor
@EqualsAndHashCode
public class Client {
    @Id
    @GeneratedValue
    public Long id;
    public String time;
}
```

Как видите через аннотации предоставили геттеры и сеттеры, конструктор и equals / hashCode. Здесь используются аннотации из Spring Data (точнее, JPA). Этот класс описывает модель с двумя полями, одно из которых генерится автоматически. По этому классу будет автоматически создана модель данных (таблицы) в БД.

5. Создаем репозиторий



```
import by.patsei.springproject1.model.Client;
```

```
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ClientRepository
    extends CrudRepository<Client, Long> {

}
```

Мы просто определяем **интерфейс** и внезапно он начинает работать с базой. Благодаря *Spring Boot* и *Spring Data* "под капотом" происходит следующее:

- 1) Увидев в зависимостях *H2* (встраиваемая БД), *Boot* автоматически конфигурирует *DataSource* (это ключевой компонент для подключения к базе) чтобы приложение работало с этой базой
- 2) *Spring Data* ищет всех наследников *CrudRepository* и автоматически генерирует для них реализацию по умолчанию, которые включают базовые методы репозитория, типа *findOne*, *findAll*, *save* etc.
- 3) *Spring* автоматически конфигурирует слой для доступа к данным

Благодаря аннотации *@Repository* этот компонент становится доступным в приложении

6. Используем репозиторий

Чтобы использовать репозиторий в контроллере воспользуемся механизмом **внедрения зависимостей (DI)**, предоставляемый *Spring Framework*. Чтобы это сделать, нужно объявить зависимость в контроллере. Увидев в конструкторе параметр типа *clientRepository*, *Spring* найдет созданный *Spring Data*-ой репозиторий и передаст его в конструктор.

```
@Controller
public class IndexController {

    final ClientRepository cRepository;

    public IndexController(ClientRepository cRepository) {
        this.cRepository = cRepository;
    }

}
```

7. Теперь пишем в базу данных

```
8. @GetMapping("/")
    public ModelAndView index() {
```

```
        ModelAndView modelAndView = new ModelAndView("index");
```



```

        modelAndView.addObject("name", "ПОИТ 3 курс");

        Client client = new Client();
        client.time = String.format("Visited at %s",
            LocalDateTime.now());
        cRepository.save(client);

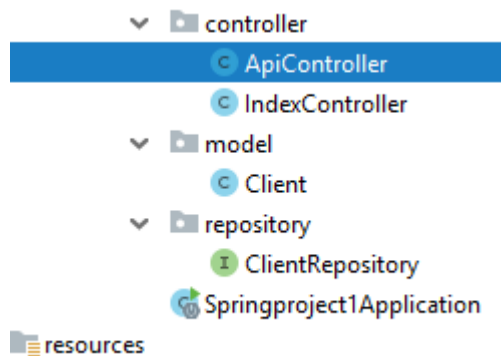
        return modelAndView;
    }

```

9. REST контроллер

Следующий шаг — надо вернуть все записи из базы в JSON формате, чтобы потом их можно было читать

REST (сокращение от *Representational State Transfer* — «передача состояния представления») — архитектурный стиль взаимодействия компонентов распределённого приложения в сети. Для REST в Spring есть отдельный тип контроллера который называется `@RestController`, код не сильно отличается от обычного контроллера.



```

import by.patsei.springproject1.model.Client;
import by.patsei.springproject1.repository.ClientRepository;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

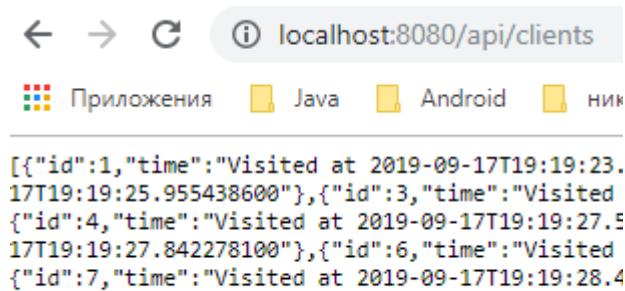
@RestController
@RequestMapping("/api")
public class ApiController {
    final ClientRepository cRepository;
    public ApiController(ClientRepository cRepository) {
        this.cRepository = cRepository;
    }
    @GetMapping("/clients")
    public Iterable<Client> getTimes() {
        return cRepository.findAll();
    }
}

```

Определяем "префикс" для всех методов контроллера `@RequestMapping`

Внедрение зависимостей работает точно так же, как и для обычных контроллеров.

Метод возвращает модель. Spring автоматически преобразует это в массив JSON объектов и при запросе `http://localhost:8080/api/clients` мы получим JSON с нужными данными.



```
{
  "id": 1, "time": "Visited at 2019-09-17T19:19:23.17719:19:25.955438600"},
  {"id": 3, "time": "Visited at 2019-09-17T19:19:27.517719:19:27.842278100"},
  {"id": 6, "time": "Visited at 2019-09-17T19:19:28.417719:19:28.695438600"},
  {"id": 7, "time": "Visited at 2019-09-17T19:19:28.695438600"}
}
```

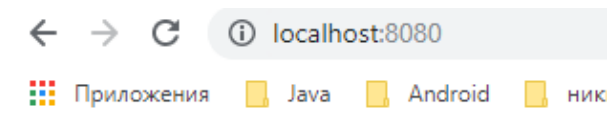
Допишем вывод списка посещений на странице



```
<!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>

<script type="text/javascript">
  fetch('/api/clients')
    .then(function(response) {
      return response.json();
    })
    .then(function(clients) {
      clients.forEach(function(visit) {
        var el = document.createElement('li');
        el.innerText = visit.time;
        document.querySelector('#clients').append(el);
      });
    });
</script>
<body>
<h1>
  Welcome to Spring,
  <h1 th:text="${name}"/>
</h1>
<ul id="clients"></ul>
</body>
</html>
```

Результат видим на экране

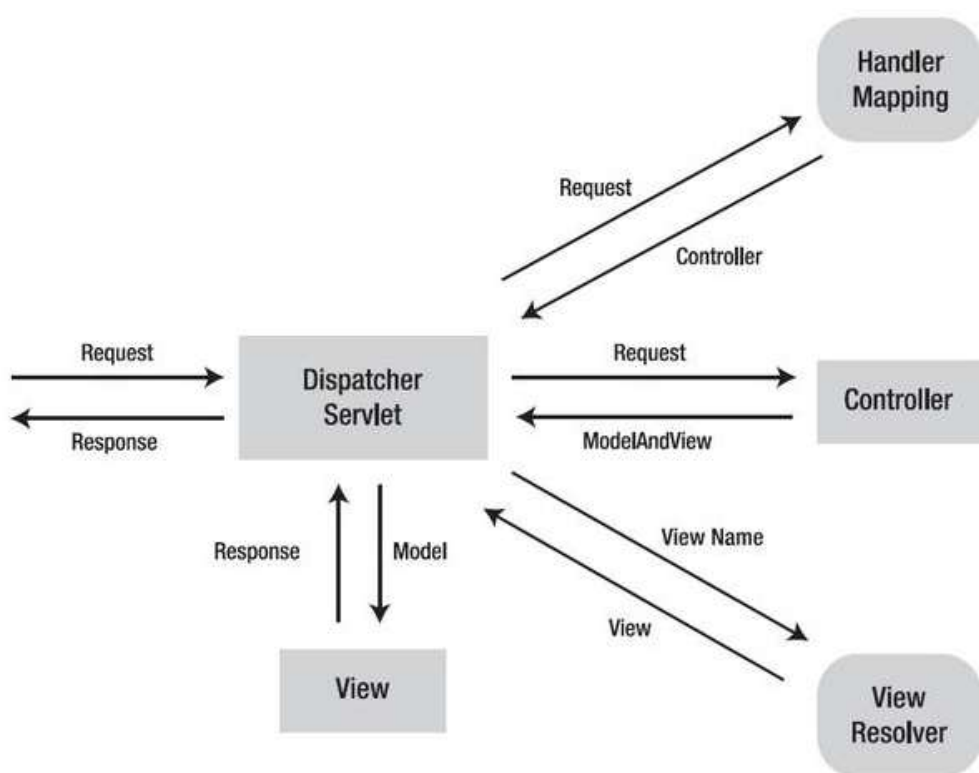


Welcome to Spring,

ПОИТ 3 курс

- Visited at 2019-09-17T13:55:55.480697400
- Visited at 2019-09-17T13:56:02.031391700
- Visited at 2019-09-17T13:56:02.843166100
- Visited at 2019-09-17T13:56:03.510372500
- Visited at 2019-09-17T13:56:04.014319400
- Visited at 2019-09-17T13:56:04.264182700
- Visited at 2019-09-17T13:56:04.545058400

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



У **Spring MVC** есть *DispatcherServlet*. Это главный контроллер, все входящие запросы проходят через него и он уже дальше передает их конкретному контроллеру. 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'a**, после чего на это

представление передаются данные модели и на выход мы получаем нашу страничку.

Аннотация **@Controller** как раз и сообщает *Spring MVC*, что данный класс является контроллером, диспетчер будет проверять аннотации **@RequestMapping** чтобы вызвать подходящий метод.

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.

Определение Configuration

Углубляемся в Spring. Рассмотрим Конфигурацию.

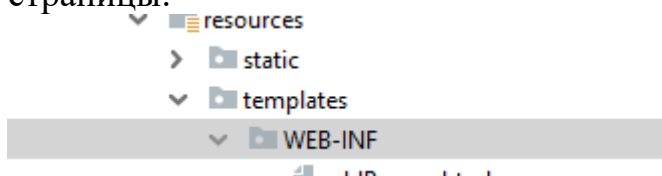
Вернемся к предыдущему проекту к классу SpringProject1Application:

```
@SpringBootApplication

public class SpringProject1Application {
    public static void main(String[] args) {
        SpringApplication.run(SpringProject1Application.class, args);
    }
}
```

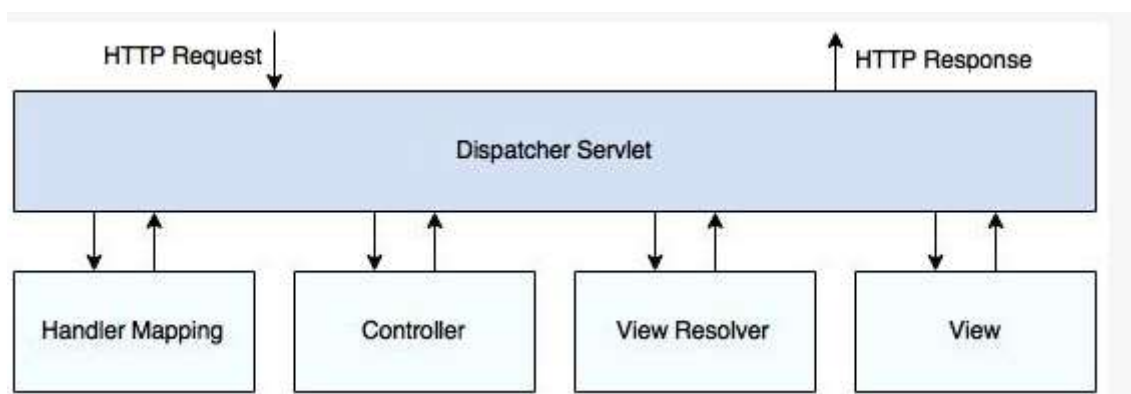
Каталог шаблонов по умолчанию - это src / main / resources / templates.

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 содержимым:

```
@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 «/».

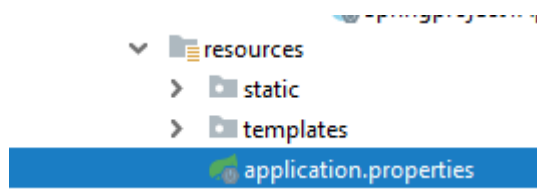
Допишите в application.properties

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

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 логирования, логи будут писаться как в консоль так и в файл одновременно.

```
#создаст файл app.log в папке temp
logging.file=${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

welcome.message= Hello Spring
error.message= First Name & Last Name is required!
```



```

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

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

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

```

@Slf4j
@Controller
public class IndexController {

```

@Slf4j, представляет собой аннотацию, предоставленную Lombok, которая во время выполнения автоматически генерирует SLF4J (Simple Logging Facade для Java, <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(IndexController.class);

```

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

```

@Slf4j
@Controller
public class IndexController {

    final ClientRepository cRepository;

    public IndexController(ClientRepository cRepository) {
        this.cRepository = cRepository;
    }

    @GetMapping("/")
    public ModelAndView index() {

        ModelAndView modelAndView = new ModelAndView("index");
        modelAndView.addObject("name", "ПОИТ 3 курс");

        Client client = new Client();

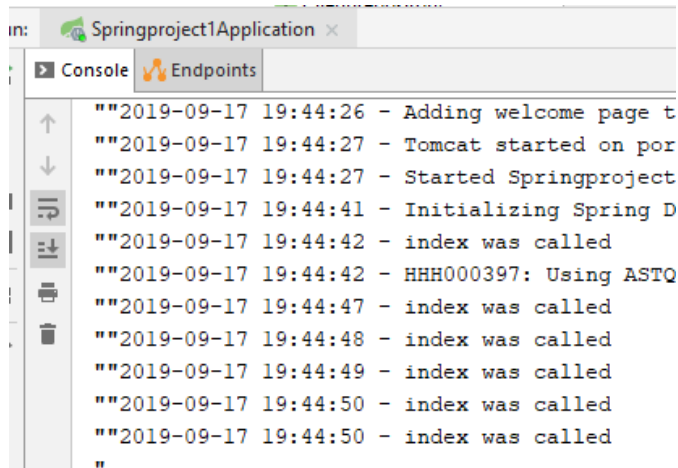
```

```

        client.time = String.format("Visited at %s", LocalDateTime.now());
        cRepository.save(client);
        log.info("index was called");
        return ModelAndView();
    }
}

```

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

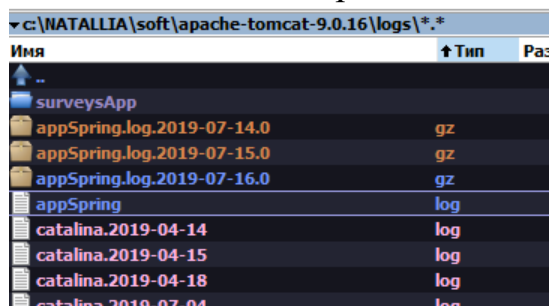


```

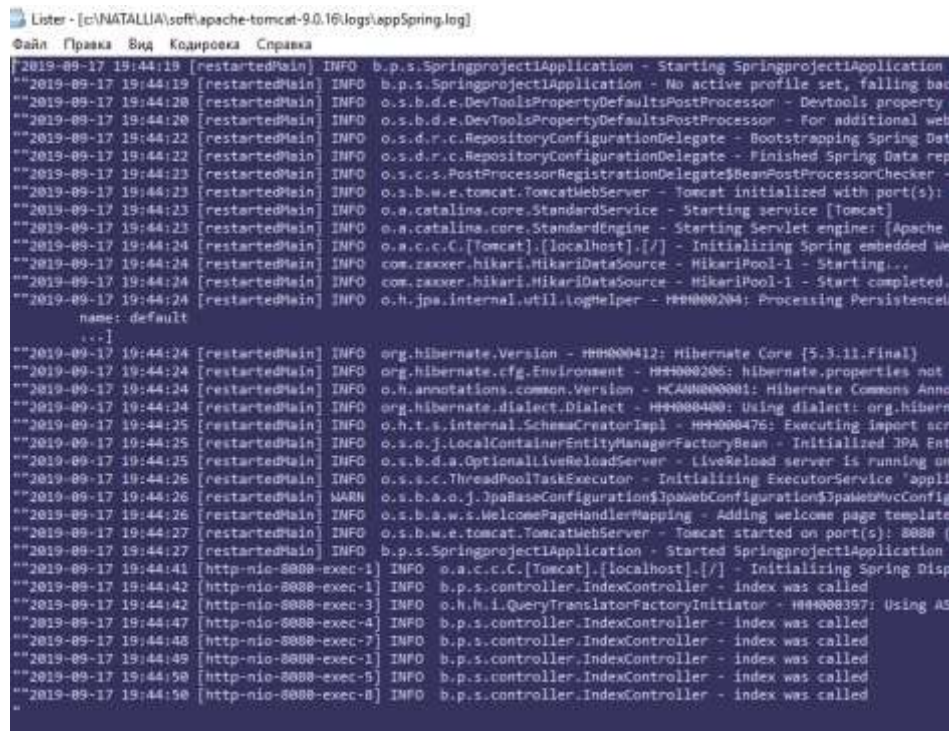
""2019-09-17 19:44:26 - Adding welcome page t
""2019-09-17 19:44:27 - Tomcat started on por
""2019-09-17 19:44:27 - Started Springproject
""2019-09-17 19:44:41 - Initializing Spring D
""2019-09-17 19:44:42 - index was called
""2019-09-17 19:44:42 - HHH000397: Using ASTQ
""2019-09-17 19:44:47 - index was called
""2019-09-17 19:44:48 - index was called
""2019-09-17 19:44:49 - index was called
""2019-09-17 19:44:50 - index was called
""2019-09-17 19:44:50 - index was called
"

```

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



Имя	Тип	Раз
..		
surveysApp		
appSpring.log.2019-07-14.0	gz	
appSpring.log.2019-07-15.0	gz	
appSpring.log.2019-07-16.0	gz	
appSpring	log	
catalina.2019-04-14	log	
catalina.2019-04-15	log	
catalina.2019-04-18	log	
catalina.2019-07-04	log	



```

2019-09-17 19:44:19 [restartedMain] INFO b.p.s.Springproject1Application - Starting Springproject1Application
""2019-09-17 19:44:19 [restartedMain] INFO b.p.s.Springproject1Application - No active profile set, falling back
""2019-09-17 19:44:20 [restartedMain] INFO o.s.b.d.e.DevToolsPropertyDefaultsPostProcessor - DevTools property
""2019-09-17 19:44:20 [restartedMain] INFO o.s.b.d.e.DevToolsPropertyDefaultsPostProcessor - For additional web
""2019-09-17 19:44:22 [restartedMain] INFO o.s.d.r.c.RepositoryConfigurationDelegate - Bootstrapping Spring Dat
""2019-09-17 19:44:22 [restartedMain] INFO o.s.d.r.c.RepositoryConfigurationDelegate - Finished Spring Data rep
""2019-09-17 19:44:23 [restartedMain] INFO o.s.c.s.PostProcessorRegistrationDelegate$BeanPostProcessorChecker -
""2019-09-17 19:44:23 [restartedMain] INFO o.s.b.w.e.tomcat.TomcatWebServer - Tomcat initialized with port(s):
""2019-09-17 19:44:23 [restartedMain] INFO o.a.catalina.core.StandardService - Starting service [Tomcat]
""2019-09-17 19:44:23 [restartedMain] INFO o.a.catalina.core.StandardEngine - Starting Servlet engine: [Apache
""2019-09-17 19:44:24 [restartedMain] INFO o.a.c.c.C.[Tomcat].[localhost].[/] - Initializing Spring embedded Web
""2019-09-17 19:44:24 [restartedMain] INFO com.zaxxer.hikari.HikariDataSource - HikariPool-1 - Starting...
""2019-09-17 19:44:24 [restartedMain] INFO com.zaxxer.hikari.HikariDataSource - HikariPool-1 - Start completed.
""2019-09-17 19:44:24 [restartedMain] INFO o.h.jpa.internal.util.LogHelper - HHH000204: Processing PersistenceC
name: default
...
""2019-09-17 19:44:24 [restartedMain] INFO org.hibernate.Version - HHH000412: Hibernate Core [5.3.11.Final]
""2019-09-17 19:44:24 [restartedMain] INFO org.hibernate.cfg.Environment - HHH000206: hibernate.properties not
""2019-09-17 19:44:24 [restartedMain] INFO org.hibernate.annotations.common.Version - HCANN000001: Hibernate Commons Anno
""2019-09-17 19:44:24 [restartedMain] INFO org.hibernate.dialect.Dialect - HHH000400: Using dialect: org.hibern
""2019-09-17 19:44:25 [restartedMain] INFO o.h.t.s.internal.SchemaCreatorImpl - HHH000476: Executing import scr
""2019-09-17 19:44:25 [restartedMain] INFO o.s.o.j.LocalContainerEntityManagerFactoryBean - Initialized JPA Ent
""2019-09-17 19:44:25 [restartedMain] INFO o.s.b.d.a.OptionalLiveReloadServer - (liveReload server is running on
""2019-09-17 19:44:26 [restartedMain] INFO o.s.s.c.ThreadPoolTaskExecutor - Initializing ExecutorService 'appli
""2019-09-17 19:44:26 [restartedMain] WARN o.s.b.a.o.j.JpaBaseConfiguration$JpaWebConfiguration$JpaWebMvcConfig
""2019-09-17 19:44:26 [restartedMain] INFO o.s.b.a.w.s.WelcomePageHandlerMapping - Adding welcome page template
""2019-09-17 19:44:27 [restartedMain] INFO o.s.b.w.e.tomcat.TomcatWebServer - Tomcat started on port(s): 8080 (
""2019-09-17 19:44:27 [restartedMain] INFO b.p.s.Springproject1Application - Started Springproject1Application
""2019-09-17 19:44:41 [http-nio-8080-exec-1] INFO o.a.c.c.C.[Tomcat].[localhost].[/] - Initializing Spring Disp
""2019-09-17 19:44:42 [http-nio-8080-exec-1] INFO b.p.s.controller.IndexController - index was called
""2019-09-17 19:44:42 [http-nio-8080-exec-3] INFO o.h.h.i.QueryTranslatorFactoryInitiator - HHH000397: Using AS
""2019-09-17 19:44:47 [http-nio-8080-exec-4] INFO b.p.s.controller.IndexController - index was called
""2019-09-17 19:44:48 [http-nio-8080-exec-7] INFO b.p.s.controller.IndexController - index was called
""2019-09-17 19:44:49 [http-nio-8080-exec-1] INFO b.p.s.controller.IndexController - index was called
""2019-09-17 19:44:50 [http-nio-8080-exec-5] INFO b.p.s.controller.IndexController - index was called
""2019-09-17 19:44:50 [http-nio-8080-exec-8] INFO b.p.s.controller.IndexController - index was called

```

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 IndexController {

    @GetMapping(value = {"/", "/index"})
    public ModelAndView index(Model model) {
        ...
    }

    @GetMapping(value = "/clientlist")
    public ModelAndView clientList(Model model) {
        ...
    }
}
```

```
@GetMapping(value = {"/addClient"})
public ModelAndView showAddClientPage(Model model) {
...
@PostMapping(value = {"/addClient"})
public ModelAndView saveClient(Model model, //
...
```

Напишите код. Запустите приложение. Проверьте все переходы.