Java для молодых специалистов Банка ВТБ

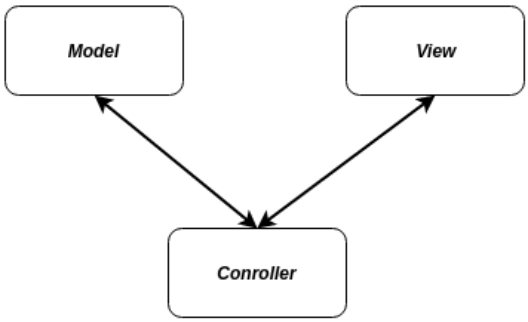
Урок 14

Spring Boot. Thymeleaf. Spring Security

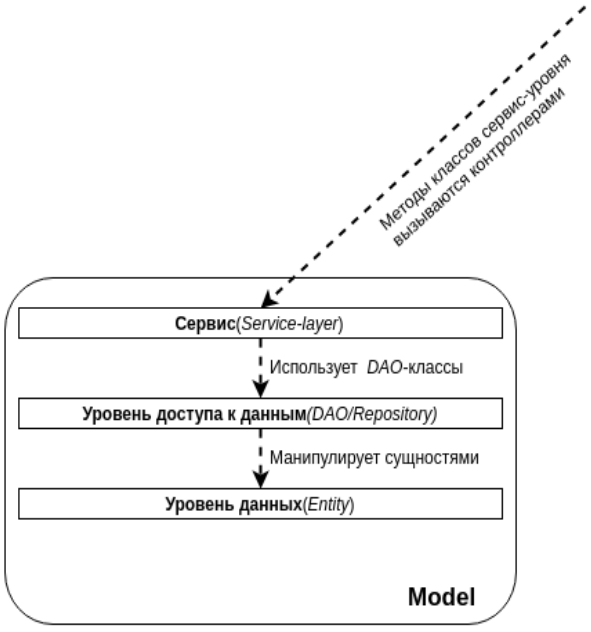
© ВТБ совместно с GeekBrains

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.

Веб-уровень же включает в себя блоки View и Controller. Конечная цель разработки веб-уровня — возможность взаимодействия пользователей с приложением непосредственно через браузер. Но кроме непосредственной реализации веб-уровня необходимо разработать соответствующую инфраструктуру, чтобы любой пользователь мог взаимодействовать с приложением через http/https-протокол.

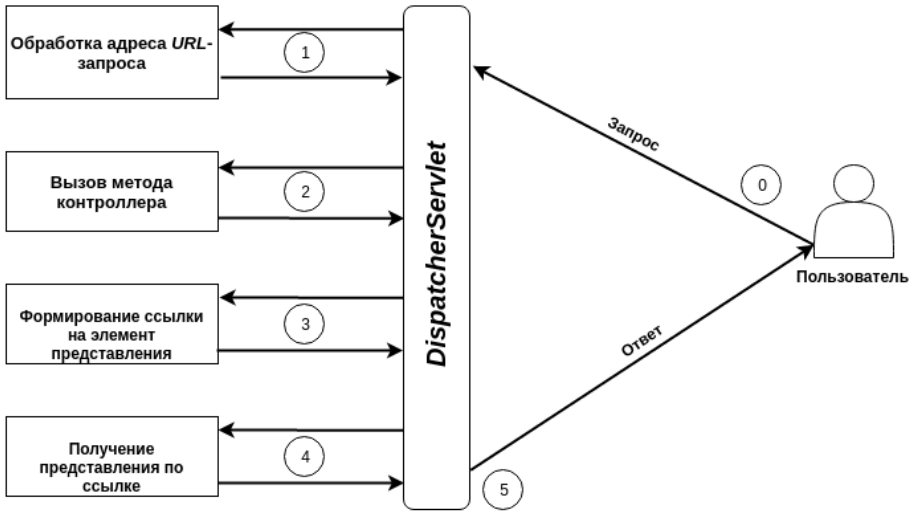
Для сетевого взаимодействия с пользователем используются классы, расширяющие интерфейс **Servlet** и его дочерние классы. Такой класс способен взаимодействовать с пользователем по принципу «запрос — ответ». Но то, как обращаться с классами-сервлетами, «знает» только контейнер

© ВТБ совместно с GeekBrains

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:



- **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) 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.

© ВТБ совместно с GeekBrains

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, 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/view/home.html**. Возврат имени производится методом **hello**, который аннотирован **@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";
  }
}
```

© ВТБ совместно с GeekBrains

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>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <title>Home</title>
    </head>

    <body>
        <h1 th:text="'Hello, ' + ${name}"></h1>
    </body>
</html>
```

В приведенном выше фрагменте html-страницы следует обратить внимание на запись (th:text = 'Hello,' + ${name}). html-страница будет ожидать передачи параметра с названием **name** из метода **hello**. Такая возможность появляется благодаря использованию шаблонизатора Thymeleaf, о котором речь пойдет дальше. Тогда метод контроллера будет выглядеть так:

```
@Controller
@RequestMapping("/home")
public class HomeController {
  @RequestMapping(value="/start", method = RequestMethod.GET)
  public String hello(Model model){
    model.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 JSP-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 model, @PathVariable(value="name") String name) {
```

© ВТБ совместно с GeekBrains

```
    model.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 model){
    List<Author> authors = authorsService.getAll();
    model.addAttribute("authors", authors);
    return "home";
  }
}
```

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

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <title>Home</title>
    </head>

    <body>
        <p th:each="author : ${authors}" th:text="${author.name}">
        </p>
    </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:

© ВТБ совместно с GeekBrains

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

  @Autowired
  public AuthorsService 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.jsp** и перейдя по адресу http://localhost:8080/app/authors/1, мы получим имя автора с id=1. Возможность получить значение из пути запроса позволяет реализовывать **REST API**. Кроме того, можно передавать возвращаемый результат напрямую в теле ответа:

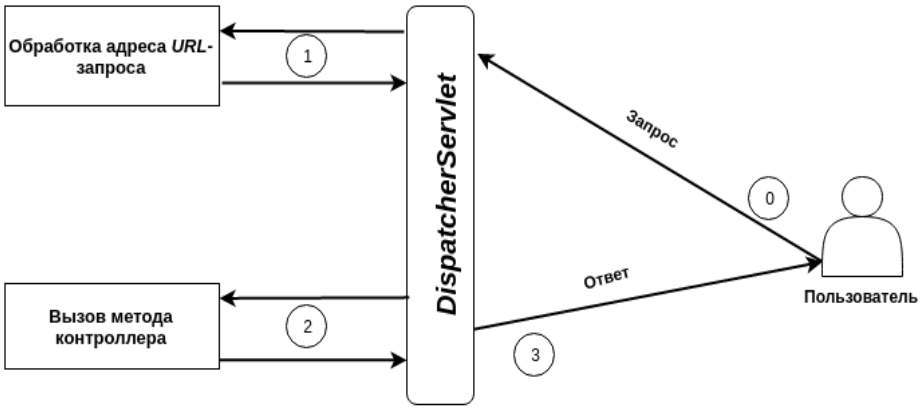
```
@Controller
@RequestMapping("/authors")
public class AuthorsController {
  private AuthorsService authorsService;

  @Autowired
  public AuthorsService 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:

© ВТБ совместно с GeekBrains



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>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <title>Home</title>
    </head>

    <body>
        <form th:action="@{/authors/add}" method="post" th:object="${author}">
            <label>
                First name
            </label>
            <input th:field="*{name}"/>
            </p>
            <label>
```

© ВТБ совместно с GeekBrains

```
                Email
            </label>
            <input th:field="*{email}"/>
            <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 JSP-страницы необходимо выполнить следующие шаги:

- **Шаг 1.** Создать пустой объект-сущность, добавить его в объект модели и возвратить имя представления, в котором содержится данная форма. Тогда **DispatcherServlet** добавит объект не по имени переменной **EL**, а по наименованию, указанному в атрибуте **modelAttribute** 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("/author")
public class AuthorsController {
  private AuthorsService authorsService;

  @Autowired
  public AuthorsService setAuthorsService(AuthorsService authorsService) {
    this.authorsService = authorsService;
  }

  @GetMapping("/author")
  public String getForm(Model model){
    Author author = new Author();
    model.addAttribute("author", author);
    return "home";
  }

  @PostMapping("/author")
  public String addAuthor(Author author) {
    authorsService.save(author);
    return "home";
  }
}
```

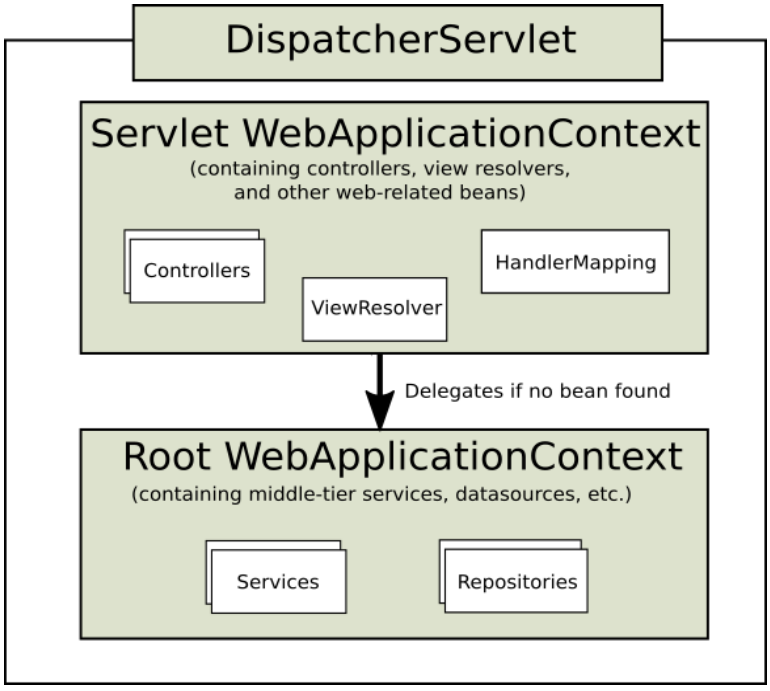
© ВТБ совместно с GeekBrains

Метод **getForm(...)** делает все, что описано в шаге 1, а метод **addAuthor(...)** соответствует шагу 2 и 3. Метод **addAuthor(...)** вызывается после того, как клиент заполнил все поля формы и отправил ее POST-запросом. В метод **addAuthor(...)** **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.

Thymeleaf

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

© ВТБ совместно с GeekBrains

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

Thymeleaf обеспечивает интеграцию со Spring Framework. Из коробки Thymeleaf позволяет обрабатывать шесть режимов шаблонов: HTML, XML, TEXT, JAVASCRIPT, CSS, RAW. По умолчанию используется режим HTML.

Диалекты Thymeleaf

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

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

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

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

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

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

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

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
...
<table>
  <thead>
    <tr>
      <th th:text="#{msgs.headers.name}">Name</th>
      <th th:text="#{msgs.headers.price}">Price</th>
    </tr>
```

© ВТБ совместно с GeekBrains

```
  </thead>
  <tbody>
    <tr th:each="prod: ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="${#numbers.formatDecimal(prod.price, 1, 2)}">0.99</td>
    </tr>
  </tbody>
</table>
```

Интеграция со Spring

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

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

Для подключения Thymeleaf к Spring Boot проекту необходимо добавить соответствующий стартер:

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

Кроме библиотеки стартер добавляет небольшой «мешочек магии» в виде класса ThymeleafAutoConfiguration, и это избавляет нас написания вручную бинов конфигурации, рассмотренных выше. А класс ThymeleafProperties определяет по умолчанию следующие опции, конфигурируемые в файле **application.properties**:

```
# THYMELEAF (ThymeleafAutoConfiguration)
spring.thymeleaf.cache=true                    # кеширование

# Check template exists before rendering it.
spring.thymeleaf.check-template=true

# Check templates location exists.
spring.thymeleaf.check-template-location=true
spring.thymeleaf.content-type=text/html        # Content-Type value.

# Enable MVC Thymeleaf view resolution.
spring.thymeleaf.enabled=true
spring.thymeleaf.encoding=UTF-8                # Template encoding.

# список отключенных представлений
spring.thymeleaf.excluded-view-names=
spring.thymeleaf.mode=HTML5                    # режим шаблонов
```

© ВТБ совместно с GeekBrains

```
spring.thymeleaf.prefix=classpath:/templates/ # путь к шаблонам, Prefix

# расширение файлов шаблонов, Suffix
spring.thymeleaf.suffix=.html
spring.thymeleaf.template-resolver-order=     # порядок поиска в цепочке

# закрытый список используемых шаблонов
spring.thymeleaf.view-names=
```

Отображение строк из файлов message.properties и интернационализация

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

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

Отметим, что Spring Boot выполняет данное конфигурирование самостоятельно.

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

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

Строки сообщений могут быть параметризированы:

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

Файл **messages.properties**:

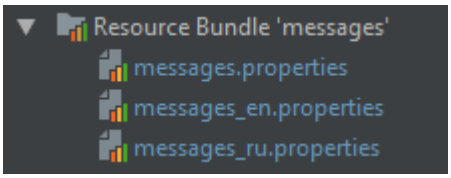
```
welcome.message=Welcome {0}!
```

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

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

Интернационализация достигается добавлением суффикса языкового кода к имени файла свойств:

© ВТБ совместно с GeekBrains



Файл **messages_ru.properties**:

```
welcome.message=Добро пожаловать {0}!
```

Отображение атрибутов модели

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

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

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

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

Отображение атрибутов коллекций

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

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

Теперь определим список новостей как атрибут модели в классе контроллера:

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

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

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

© ВТБ совместно с GeekBrains

Форматированный вывод

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

```
<td th:text="${{item.date}}"/>
```

И определить бин пользовательского форматтера в конфигурации приложения:

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

```
public class DateFormatter implements Formatter<Date> {
  @Autowired
  private MessageSource messageSource;

  public Date parse(final String text, final Locale locale)
                      throws ParseException {
    final SimpleDateFormat dateFormat = createDateFormat(locale);
    return dateFormat.parse(text);
  }

  public String print(final Date object, final Locale locale) {
    final SimpleDateFormat dateFormat = createDateFormat(locale);
    return dateFormat.format(object);
  }

  private SimpleDateFormat createDateFormat(final Locale locale) {
    final String format = this.messageSource.getMessage("date.format",
                                                        null, locale);
    final SimpleDateFormat dateFormat = new SimpleDateFormat(format);
    dateFormat.setLenient(false);
    return dateFormat;
  }
}
```

Обработка форм

Действие формы может быть указано с помощью атрибута **th:action**. Атрибут **th:object** связывает данную форму с указанным объектом модели. Отдельные поля отображаются с использованием атрибута **th:field="*{name}"**, где **name** — имя свойства объекта (по конвенции вызывается соответствующий геттер).

Следующий пример шаблона:

```
<form action="#" th:action="@{/some}" th:object="${item}" method="post">
```

© ВТБ совместно с GeekBrains

```
  <input type="text" th:field="*{date}" />
  <input type="text" th:field="*{header}" />
  <input type="text" th:field="*{text}" />
</form>
```

… отображается в:

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

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

Условные выражения

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

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

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

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

Пример:

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

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

Spring Boot

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

© ВТБ совместно с GeekBrains

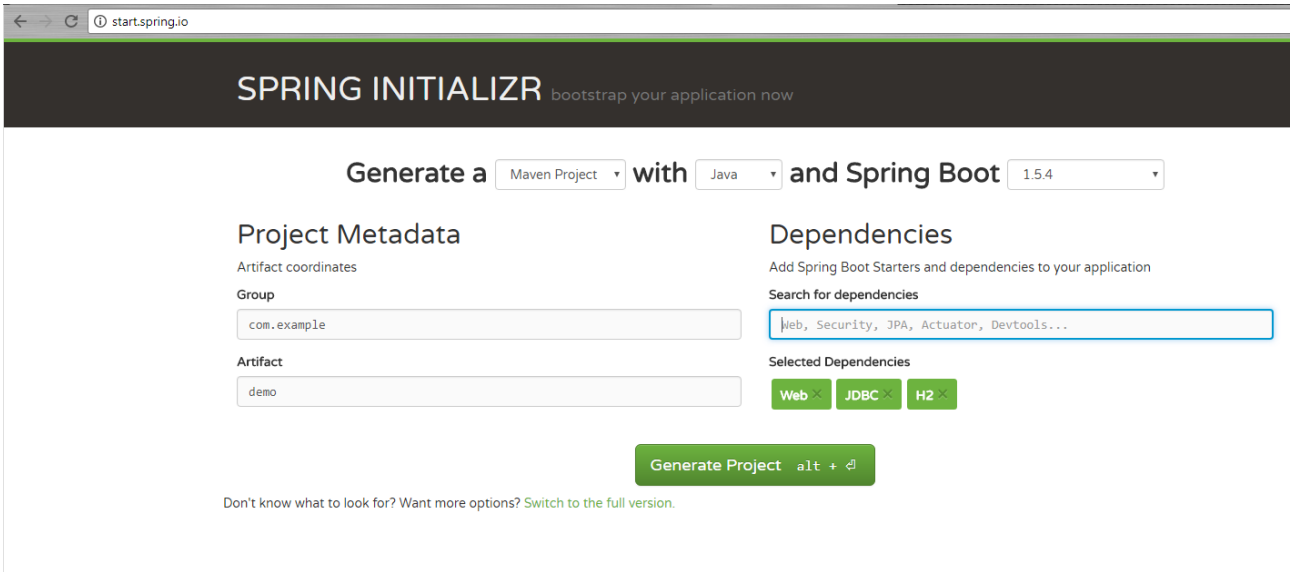
Цели, для которых создали Spring Boot

- Быстрое начало разработки и широкое применение накопленного опыта.
- Самодостаточность в работоспособности базовых настроек и легкая смена поведения при необходимости.
- Предоставление крупных готовых строительных блоков, используемых в широком классе применений, в том числе: различные встроенные серверы, безопасность, показатели, проверки работоспособности, внешняя конфигурация.
- Отсутствие генерируемого кода и необходимости XML-конфигурирования.

Быстрый старт

start.spring.io

Start.spring.io — это веб-сервис, призванный построить каркас приложения с определенной структурой каталогов и файлов, которая понятна современным средствам автоматизированной сборки проектов (например, Gradle или Maven). В этом уроке рассмотрим структуру Maven-проекта.



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

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

© ВТБ совместно с GeekBrains

6. Папка **src/test** — исходные файлы для автоматического тестирования.

7. Папка **src/test/java** — исходные файлы Java для автоматического тестирования.

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

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

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

		<dependency>
			<groupId>com.h2database</groupId>
			<artifactId>h2</artifactId>
			<scope>runtime</scope>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
	</dependencies>
	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>
</project>
```

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

© ВТБ совместно с GeekBrains

В итоге все очень быстро и удобно, не надо вручную писать («копипастить») **pom.xml**. И вы всегда получаете самые свежие стабильные версии Spring-зависимостей — не нужно думать о номере текущей актуальной версии фреймворка или его дополнений.

Spring Boot CLI

Кроме описанных выше вариантов для поклонников командной строки существует **Spring Boot CLI**. Одна из его функций — инициализация нового проекта напрямую из командной строки:

```
> spring init -d=h2,web,jdbc,thymeleaf  -n demo -x
```

Данная команда также исполнит HTTP-запрос к сервису https://start.spring.io, получит .zip-файл, и, благодаря ключу **-x**, распакует его в вышеописанную структуру в текущей папке. По умолчанию используется jar-формат и стандартный boot-стартер, проекту будет присвоено имя *demo*.

Быстрая разработка приложений со Spring Boot

Из стартового набора рассмотрим еще два файла:

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

```
@RestController
@SpringBootApplication
public class DemoApplication {
  public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);
  }

  @RequestMapping("/**")
  public String helloWorld() {
    return "Hello World";
  }
}
```

У класса **DemoApplication** двойное назначение:

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

Атрибут **@SpringBootApplication** в числе прочих наследуется от атрибутов **@Configuration**, **@ComponentScan**, **@EnableAutoConfiguration**:

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

© ВТБ совместно с GeekBrains

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

Стартеры

Стартеры — это наборы удобных дескрипторов зависимостей, которые можно включить в приложение. Стартеры содержат множество зависимостей, необходимых для быстрого запуска проекта с помощью согласованного и поддерживаемого набора управляемых транзитивных зависимостей. Все официальные стартеры следуют единой схеме именования: **spring-boot-starter-***, где * — конкретный тип приложения.

Вот несколько популярных стартеров Spring Boot:

- **spring-boot-starter-web** — используется для создания веб-служб RESTful с использованием Spring MVC и Tomcat в качестве встроенного контейнера приложений;
- **spring-boot-starter-jersey** — альтернатива Spring-boot-starter-web, в которой используется встроенный сервер приложений Jersey, а не Tomcat;
- **spring-boot-starter-jdbc** — реализует пул соединений JDBC, основан на реализации пула JDBC Tomcat.

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

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

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

Указание версии Java

spring-boot-starter-parent выбирает совместимость с Java 6. Варианты Spring-инициализатора позволяют указать конкретную версию Java, с которой вы будете работать, что в итоге выливается в следующие строки pom-файла:

```
<properties>
  <java.version>1.8</java.version>
</properties>
```

Spring Boot Maven plugin

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

© ВТБ совместно с GeekBrains

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

Данный плагин решает несколько полезных задач:

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

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

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

```
<packaging>war</packaging>
```

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

Свойства (application.properties)

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

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

```
server.port=8189
```

Spring Boot, зная, что вы используете реализацию PostgreSQL JDBC-драйвера, сам создает бины DataSource, свойства которых выставляет по умолчанию. Чтобы выставить другие значения, их можно прописать в файле свойств.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.driver-class-name=org.postgresql.Driver
```

© ВТБ совместно с GeekBrains

Особенности различных применений

Статический контент

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

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

```
spring.mvc.static-path-pattern=/resources/**
```

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

Spring Security

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

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

Конфигурирование

Для начала необходимо подключить к проекту следующие зависимости:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>5.0.7.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>5.0.7.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
```

© ВТБ совместно с GeekBrains

```
    <artifactId>thymeleaf-extras-springsecurity5</artifactId>
    <version>3.0.3.RELEASE</version>
</dependency>
```

Первые две зависимости представляют собой непосредственно модуль Spring Security, а третья — библиотеку для интеграции Thymeleaf и Spring Security. Версии Spring MVC и Spring Security «не идут параллельно» и могут отличаться — необходимо по документации проверять их совместимость. В данном случае используются Spring MVC 5.0.8 и Spring Security 5.0.7.

Рассмотрим конфигурирование Spring Security через JavaConfig — для этого добавим в проект два класса: **SecurityWebApplicationInitializer** и **SecurityConfig**. Spring Security предоставляет фильтры для защиты на уровне запросов. Чтобы подключить их к проекту, достаточно создать класс **SecurityWebApplicationInitializer** с наследованием от **AbstractSecurityWebApplicationInitializer**.

Код класса **SecurityWebApplicationInitializer**:

```
@Component
public class SecurityWebApplicationInitializer extends
AbstractSecurityWebApplicationInitializer {
}
```

Чтобы настроить правила защиты приложения, применяется класс конфигурации **SecurityConfig**, наследуемый от **WebSecurityConfigurerAdapter**.

Код класса **SecurityConfig**:

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    private DataSource dataSource;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
{ // (1)
        auth.jdbcAuthentication().dataSource(dataSource);
    }

//    @Override
//    protected void configure(AuthenticationManagerBuilder auth) throws
Exception { // (2)
//        User.UserBuilder users = User.withDefaultPasswordEncoder();
//        auth.inMemoryAuthentication()
//
```

```
.withUser(users.username("user1").password("pass1").roles("USER", "ADMIN"))
//
.withUser(users.username("user2").password("pass2").roles("USER"));
//    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/").hasAnyRole("USER")
                .antMatchers("/admin/**").hasRole("ADMIN")
                  .and()
                .formLogin()
                  .loginPage("/login")
                  .loginProcessingUrl("/authenticateTheUser")
                .permitAll();
    }
}
```

@EnableGlobalMethodSecurity позволяет использовать аннотации для защиты на уровне методов: можно на основе ролей ограничивать права доступа к отдельным методам с помощью аннотаций **@Secured** и **@PreAuthorized** (рассмотрим их позднее).

Конфигурация процесса авторизации и аутентификации определяется в методе **configure(AuthenticationManagerBuilder auth)**. В предыдущем примере кода рассмотрены два варианта настройки: первый в качестве источника данных использует БД, а второй (закомментированный) хранит информацию о пользователях просто в памяти — он приведен для примера.

- **auth.jdbcAuthentification().dataSource()** — указывает источник данных, из которого **Spring Security** будет получать информацию о пользователях и их уровнях доступа. В данном случае этот **bean** инжектится из **beans.xml**:

```
<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
destroy-method="close">
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    <property name="jdbcUrl"
value="jdbc:mysql://localhost:3306/geek_db?useSSL=false"/>
    <property name="user" value="geek"/>
    <property name="password" value="geek"/>
    <property name="minPoolSize" value="5"/>
    <property name="maxPoolSize" value="20"/>
    <property name="maxIdleTime" value="30000"/>
</bean>
```

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

© ВТБ совместно с GeekBrains

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

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

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

Подготовка базы данных

Чтобы хранить информацию о пользователях и работать с ней, применим PostgreSQL базу данных. Пример подключения ее параметров был приведен выше. По умолчанию Spring Security будет использовать стандартный шаблон таблиц в БД: пользователи хранятся в таблице **users**, а роли — в **authorities**.

В таблице **users** три столбца:

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

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

Запрос на создание такой таблицы и добавление тестовых пользователей:

```
CREATE TABLE users (
```

© ВТБ совместно с GeekBrains

```
    username varchar(50) NOT NULL,
    password varchar(100) NOT NULL,
    enabled boolean(1) NOT NULL,
    PRIMARY KEY (username)
);

INSERT INTO users
VALUES
('user1', '{noop}123', 1),
('user2', '{noop}123', 1);

CREATE TABLE authorities (
    username varchar(50) NOT NULL,
    authority varchar(50) NOT NULL,

    CONSTRAINT authorities_idx UNIQUE (username, authority),

    CONSTRAINT authorities_ibfk_1
    FOREIGN KEY (username)
    REFERENCES users (username)
);

INSERT INTO authorities
VALUES
('user1', 'ROLE_ADMIN'),
('user1', 'ROLE_USER'),
('user2', 'ROLE_USER');
```

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

Авторизация

Базовая настройка базы данных, Spring Security и пользователей выполнена. Теперь реализуем возможность авторизации на сайте. Необходимы форма и метод обработки GET-запроса для нее.

```
@GetMapping("/login")
public String showMyLoginPage() {
    return "modern-login";
}
```

Метод возвращает html-страницу с именем **modern-login.html**. Код страницы:

```
<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org" lang="en">

<head>
    <title>Login Page</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
```

© ВТБ совместно с GeekBrains

```
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
</head>

<body>

<div>
    <div id="loginbox" style="margin-top: 50px;" class="mainbox col-md-3 col-md-offset-2
col-sm-6 col-sm-offset-2">
        <div class="panel panel-info">
            <div class="panel-heading">
                <div class="panel-title">Sign In</div>
            </div>
            <div style="padding-top: 30px" class="panel-body">
                <form th:action="@{/authenticateTheUser}" method="POST"
class="form-horizontal">
                    <div class="form-group">
                        <div class="col-xs-15">
                            <div>
                                <div th:if="${param.error} != null">
                                    <div class="alert alert-danger col-xs-offset-1 col-xs-10">
                                        Invalid username or password
                                    </div>
                                </div>
                                <div th:if="${param.logout} != null">
                                    <div class="alert alert-success col-xs-offset-1 col-xs-10">
                                        You have been logged out.
                                    </div>
                                </div>
                            </div>
                        </div>
                    </div>
                    <div style="margin-bottom: 25px" class="input-group">
                        <span class="input-group-addon"><i class="glyphicon
glyphicon-user"></i></span>
                        <input type="text" name="username" placeholder="username"
class="form-control">
                    </div>
                    <div style="margin-bottom: 25px" class="input-group">
                        <span class="input-group-addon"><i class="glyphicon
glyphicon-lock"></i></span>
                        <input type="password" name="password" placeholder="password"
class="form-control">
                    </div>
                    <div style="margin-top: 10px" class="form-group">
                        <div class="col-sm-6 controls">
                            <button type="submit" class="btn btn-success">Login</button>
                        </div>
                    </div>
                </form>
            </div>
        </div>
    </div>
</div>
</body>
</html>
```

У полей для ввода логина и пароля стандартные имена: **username** и **password**. Форма посылает POST-запрос по адресу **@{/authentificateTheUser}**. Если мы попали на эту страницу после неудачной попытки авторизации или после выхода из учетной записи, на форме будут показаны соответствующие сообщения. После авторизации пользователь получает набор прав, указанный в БД, и может пользоваться веб-приложением.

Защита на уровне представлений

Рассмотрим, как изменять видимость элементов на странице в зависимости от прав пользователей. Для этого используется зависимость **thymeleaf-extras-springsecurity5** из **pom.xml**. Пример использования таких дополнительных возможностей:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<html xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
<!--...-->
<body>
<div class="container">
    <h1>Welcome page</h1>
    <div sec:authorize="isAuthenticated()">
        Authenticated username:
        <div sec:authentication="principal.username"></div>
        Authenticated user roles:
        <div sec:authentication="principal.authorities"></div>
    </div>

    <!--<div sec:authorize="hasAnyRole('ADMIN', 'USER')">-->
    <div sec:authorize="hasRole('ADMIN')">
        This content will only be visible to ADMIN users.
    </div>

    <h2>Index:</h2>
<!--...-->
</div>
</body>
</html>
```

Здесь ** ** отвечает за проверку авторизации пользователя. Если он авторизован, на странице отображаются его **username** и права доступа (например, **ROLE_ADMIN** или **ROLE_USER**). Методы **hasRole()** и **hasAnyRole()** проверяют у пользователя наличие определенной роли.

Защита на уровне методов

Последний шаг для создания надежной защиты приложения — ограничить доступ на уровне методов. Для этого можно использовать аннотацию **@Secured**, которая ограничивает доступ к отдельным методам на основе информации о правах текущего пользователя. Пример кода:

```
@Secured({"ROLE_ADMIN"})
```

© ВТБ совместно с GeekBrains

```
@RequestMapping("/onlyYou")
@ResponseBody
public String onlyYou() {
    return "index";
}
```

Доступ к методу **onlyYou()** имеют только пользователи с ролью ADMIN.

Практическое задание

Упрощенный вариант домашнего задания:

1. Разберитесь с имеющимся кодом;
2. Реализуйте удаление товаров;
3. * Добавьте input поле и кнопку “Фильтровать” на странице. По нажатию на кнопку мы должны увидеть страницу с товарами, содержащими в себе указанную подстроку. Сам фильтр при этом может сброситься.

Обычный вариант домашнего задания:

1. Добавьте возможность редактировать название и цену товаров;
2. На странице с товарами добавить фильтры по цене (минимальная, максимальная), названию (найти все товары в которых встречается указанная подстрока, про регистр можно не думать). Фильтр это просто набор input’ов, куда можно вбить ограничения. Имейте ввиду, что после выполнения фильтрации, мы должны увидеть страницу с отобранными товарами, и при этом сами фильтры сброситься не должны.

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.
2. Брюс Эккель. Философия Java // 4-е изд.: Пер. с англ. — СПб.: Питер, 2016. — 1 168 с.
3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.
4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.

© ВТБ совместно с GeekBrains