Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт космических и информационных технологий
Кафедра «Информатика»

Конспект лекции №5
Spring MVC

Составил: Старший преподаватель кафедры «Информатика»
Черниговский Алексей Сергеевич

Красноярск 2020

Содержание

1 Реализация паттерна MVC в веб-приложениях с помощью фреймворка Spring

В фреймворке Spring наряду с другими модулями есть и свой веб-фреймворк, известный под названием Spring Web MVC. В его основе лежит паттерн «Модель — Представление — Контроллер» (Model — View — Controller, MVC). Модуль Web MVC фреймворка Spring поддерживает уровень визуализации данных (presentation tier) и помогает в создании гибкого и слабо сцепленного веб-приложения. Модуль MVC фреймворка Spring решает задачу тестирования веб-компонентов в корпоративном приложении. С его помощью можно писать тестовые сценарии без использования в приложении объектов запроса/ответа.

Основной подход данного паттерна заключается в продвижении в среде разработчиков ПО принципа разделения ответственности. Паттерн MVC разделяет систему на три вида компонентов, причем у каждого компонента системы есть свои конкретные обязанности. Вот эти три вида компонентов.

Модель. Модель в паттерне MVC отвечает за данные для представления, с целью дальнейшей их визуализации в одном из шаблонов представления.

Представление. Представление в паттерне MVC отвечает за визуализацию модели в веб-приложении в виде веб-страницы. Оно представляет данные модели в удобочитаемом для пользователя формате. Для этой цели существует несколько технологий, например JSP, страницы JSF, PDF, XML и др.

Контроллер. Это тот компонент паттерна MVC, который фактически производит все действия. Код контроллера управляет взаимодействием между представлением и моделью. Такие взаимодействия, как отправка формы или щелчок на ссылке, являются в корпоративном приложении частью контроллера. Кроме того, контроллер отвечает за создание и обновление модели, а также перенаправление модели представлению для визуализации.

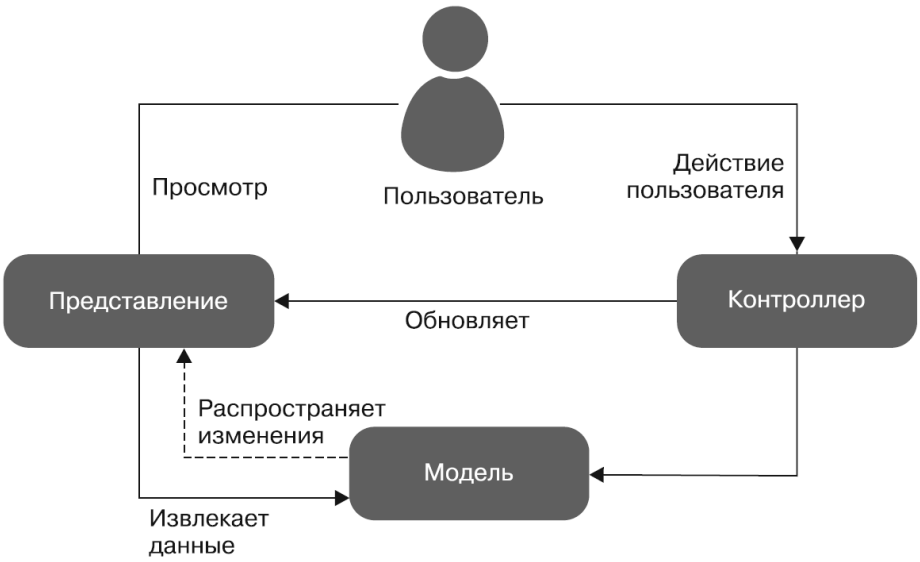

Рисунок 1 — Принцип работы паттерна MVC

Рассмотрим следующую схему, чтобы лучше разобраться в паттерне MVC (Рисунок 1).

В приложении есть три компонента, у каждого из которых — свои обязанности. Как уже упоминалось, паттерн MVC нацелен на разделение ответственности. В программной системе разделение ответственности играет очень важную роль, обеспечивая гибкость и легкость тестирования компонентов, а также чистую структуру кода. В паттерне MVC пользователь взаимодействует с компонентом «контроллер» посредством компонента «представление», а компонент «контроллер» запускает фактические действия по подготовке компонента «модель». Компонент «модель» распространяет изменения на «представление», и, наконец, компонент «представление» визуализирует модель на экране пользователя. Такова общая идея реализации паттерна MVC. Паттерн MVC отлично подходит для большинства приложений, особенно традиционных, не веб-. Паттерн MVC известен также как архитектура «Модель 1».

Впрочем, корпоративные веб-приложения несколько отличаются от традиционных приложений, ведь в силу того, что протокол HTTP не сохраняет состояние, сохранять информацию о модели в течение всего жизненного цикла запроса довольно трудно. В следующем разделе мы рассмотрим усовершенствованную версию паттерна MVC и ее применение во фреймворке Spring для создания корпоративного веб-приложения.

2 Архитектура «Модель 2» паттерна MVC в Spring

Архитектура «Модель 1» для веб-приложения представляется не очень простой. В «Модели 1» есть также децентрализованное управление навигацией по сайту, ведь в этой архитектуре у каждого пользователя есть отдельный контроллер и отдельная логика определения следующей страницы. Основные технологии для разработки веб-приложений с архитектурой «Модель 1» — сервлеты и JSP.

Сервлет представляет специальный тип классов Java, который выполняется на веб-сервере и который обрабатывает запросы и возвращает результат обработки.

Для веб-приложений паттерн MVC реализуется в виде архитектуры «Модель 2». Эта архитектура обеспечивает централизованную логику управления навигацией, что позволяет легко тестировать и сопровождать веб-приложение, и разделение обязанностей в ней для веб-приложений организовано лучше, чем в архитектуре «Модель 1». Усовершенствованный паттерн MVC на основе архитектуры «Модель 2» отличается от паттерна MVC на основе архитектуры «Модель 1» наличием контроллера — единой точки входа, распределяющего все входящие запросы по другим контроллерам. Эти контроллеры обрабатывают входящие запросы, возвращая модель, и выбирают

представление. Рассмотрим следующую схему, чтобы лучше понять принципы функционирования паттерна MVC архитектуры «Модель 2» (Рисунок 2).

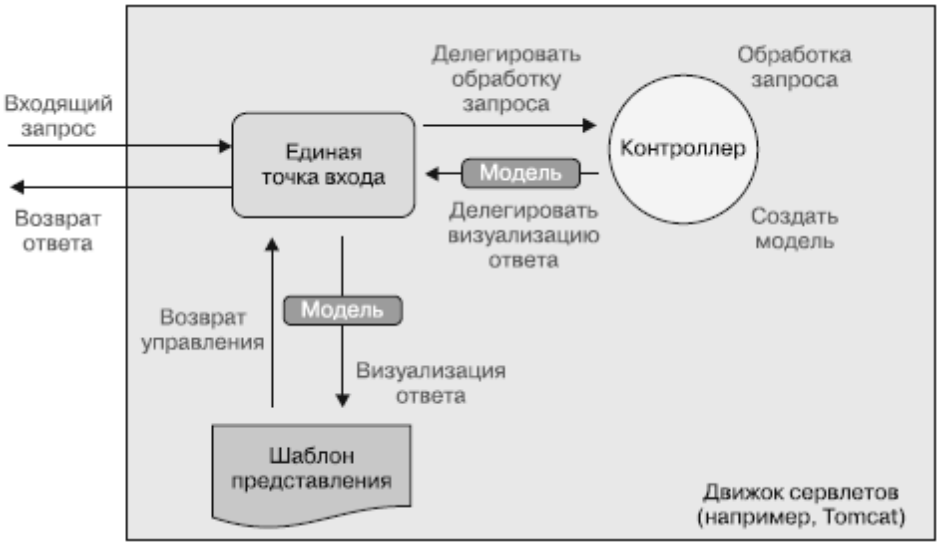

Рисунок 2 — Функционирование паттерна MVC архитектуры «Модель 2»

Как можно видеть на схеме, в паттерне MVC появляется новый компонент, а именно единая точка входа (front controller). Он реализуется в виде сервлета типа javax.servlet.Servlet, например ActionServlet во фреймворке Apache Struts, FacesServlet в JSF или DispatcherServlet в MVC Spring. Он получает входящие запросы, передавая их одному из контроллеров приложения, который, в свою очередь, создает и обновляет модель, передавая ее обратно единой точке входа для визуализации. Наконец, единая точка входа определяет конкретное представление и визуализирует данные модели.

3 Паттерн проектирования «Единая точка входа»

Паттерн проектирования «Единая точка входа» — паттерн J2EE, решающий следующие проблемы проектирования приложений.

- В основанных на архитектуре «Модель 1» веб-приложениях для обработки большого количества запросов необходимо слишком много контроллеров. Их сопровождение и переиспользование представляет собой непростую задачу.

- У каждого запроса своя точка входа в приложение. Желательно, чтобы точка входа была одна для всех запросов.

JSP и сервлеты — основные компоненты паттерна MVC с архитектурой «Модель 1», так что эти компоненты отвечают как за фактические действия, так и за визуализацию, нарушая тем самым принцип единственной обязанности.

Единая точка входа предлагает решение вышеописанных проектных проблем веб-приложения. В веб-приложении она играет роль основного компонента, маршрутизирующего все запросы. Это значит, что в отдельный контроллер (единую точку входа) попадает слишком много запросов, так что их

обработка затем делегируется конкретным контроллерам. Единая точка входа обеспечивает централизованное управление и расширяет возможности переиспользования и удобство управления, поскольку обычно в веб-контейнере регистрируется только ресурс. Эта точка входа не только распределяет/обрабатывает излишки запросов, но также отвечает за:

- инициализацию обслуживающего запросы фреймворка;
- загрузку ассоциативного массива всех URL и компонентов, отвечающих за обработку запроса;
- подготавливает ассоциативный массив для представлений.

Рассмотрим следующую схему единой точки входа (Рисунок 3).

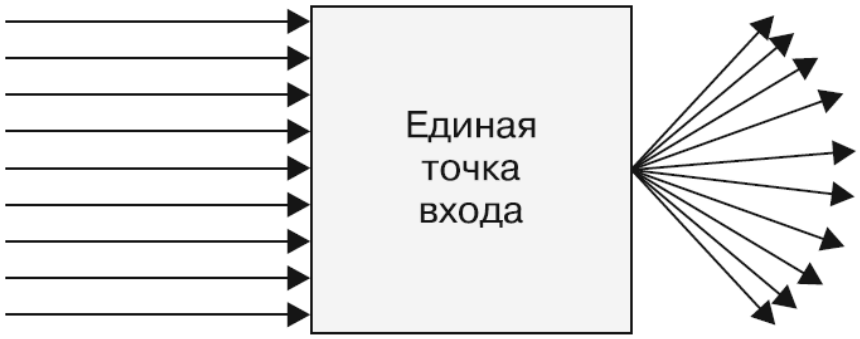

Рисунок 3 — Единая точка входа

Как можно видеть на предыдущей схеме, все запросы приложения попадают в единую точку входа, которая делегирует эти запросы предварительно настроенным контроллерам приложения.

Фреймворк Spring содержит модуль, основанный на паттерне MVC, — реализацию архитектуры «Модель 2». Модуль MVC фреймворка Spring предоставляет готовую реализацию паттерна «Единая точка входа» в виде класса org.springframework.web.servlet.DispatcherServlet. Этот простой класс сервлета составляет основу модуля MVC фреймворка Spring. И он интегрирован с контейнером IoC фреймворка Spring, что позволяет воспользоваться возможностями паттерна внедрения зависимостей Spring. Веб-фреймворк Spring использует сам Spring для настройки, а все контроллеры представляют собой компоненты Spring, причем с возможностями тестирования.

Подробнее изучим в следующей главе внутреннее устройство модуля MVC Spring и взглянем поближе на то, как класс org.springframework.web.servlet.DispatcherServlet фреймворка MVC Spring обрабатывает все входящие запросы к веб-приложению.

4 Технологический процесс жизненного цикла запроса

Изучим технологический процесс обработки запросов в модуле MVC фреймворка Spring для веб-приложений. Технологический процесс обработки

запросов класса DispatcherServlet веб-фреймворка MVC Spring показан на рисунке 4.

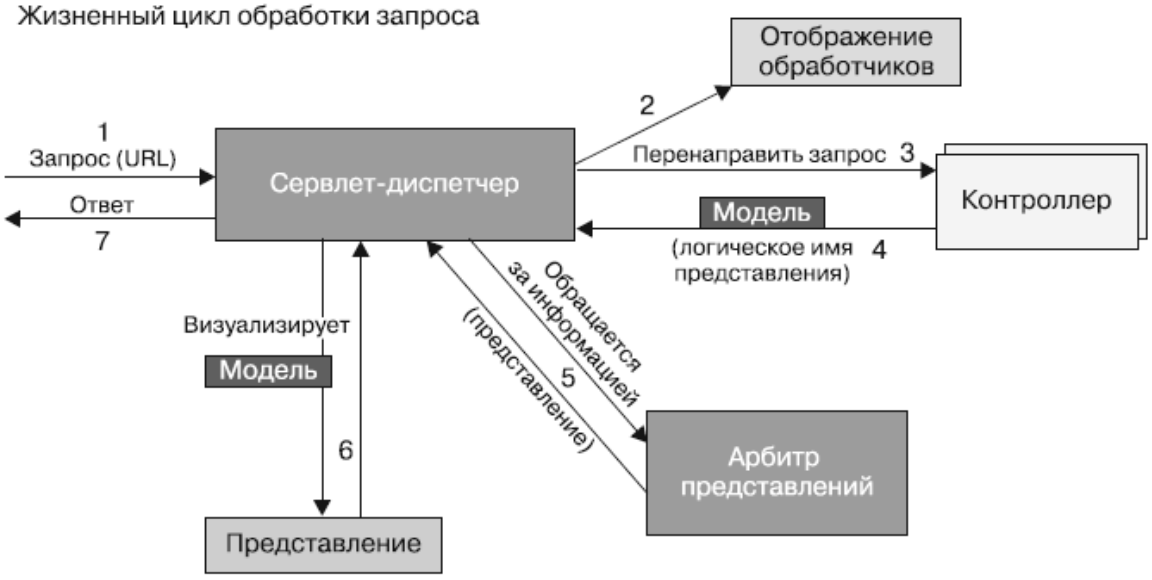

Рисунок 4 — Технологический процесс обработки запросов класса DispatcherServlet фреймворка MVC

Как вы уже знаете, единая точка входа играет очень важную роль в паттерне MVC архитектуры «Модель 2», поскольку отвечает за все входящие запросы к веб-приложению и подготовку ответа браузеру. Во фреймворке MVC Spring объект org.springframework.web.servlet.DispatcherServlet играет роль единой точки входа паттерна MVC архитектуры «Модель 2». Как вы можете видеть на рис. 4, этот объект DispatcherServlet для выполнения своих задач использует множество других компонентов. Рассмотрим поэтапно обработку запроса во фреймворке MVC Spring.

1. Пользователь нажимает на ссылку в браузере или отправляет веб-форму в приложении. Запрос, включающий или какую-то дополнительную, или просто основную информацию, покидает браузер и оказывается в объекте DispatcherServlet фреймворка Spring, представляющем собой просто класс сервлета, как и другие веб-приложения на основе Java. Он представляет собой единую точку входа фреймворка MVC Spring, через которую пропускаются все входящие запросы. Благодаря использованию этой единой точки входа фреймворк MVC Spring централизует управление всем потоком запросов.

2. После поступления запроса в объект DispatcherServlet фреймворка Spring последний передает этот запрос в контроллер MVC Spring, то есть контроллер приложения. Хотя в веб-приложениях Spring может быть несколько контроллеров, но каждый запрос должен попасть к одному из них. Для этого объект DispatcherServlet пользуется отображениями обработчиков, заданными в настройках веб-приложения. Отображение обработчиков определяет конкретный контроллер по URL и параметрам запроса.

3. После выбора нужного контроллера приложения объектом DispatcherServlet с помощью настроек отображения обработчиков DispatcherServlet направляет запрос этому контроллеру. Именно этот контроллер на самом деле отвечает за обработку информации в соответствии с запросом пользователя и его параметрами.

4. Контроллер фреймворка MVC Spring выполняет бизнес-логику, используя бизнес-сервисы приложения, и создает модель, в которую обертывается отправляемая обратно пользователю и отображаемая в браузере информация. Эта модель несет соответствующую запросу пользователя информацию, но она не форматирована, так что можно использовать любую технологию шаблонов представлений для визуализации в браузере содержащейся в ней информации. Именно поэтому контроллер MVC Spring возвращает, помимо модели, название логического представления. А делает он это потому, что контроллер MVC Spring не привязан ни к какой конкретной технологии представления — JSP, JSF, Thymeleaf и т. д.

5. И снова объект DispatcherServlet модуля MVC фреймворка Spring пользуется помощью арбитра представлений, который настроен в веб-приложении таким образом, чтобы разрешать представления. В соответствии с настройками объекта ViewResolver он выдает реальное имя представления вместо логического имени представления. Теперь у объекта DispatcherServlet есть необходимое для визуализации информации модели представление.

6. Объект DispatcherServlet модуля MVC фреймворка Spring визуализирует модель в представление и генерирует информацию модели в удобочитаемом для пользователя формате.

7. Наконец, на основе этой информации DispatcherServlet создает ответ и возвращает его браузеру пользователя.

Как видите, процесс обработки запроса приложения состоит из нескольких этапов и в него вовлечено несколько компонентов. Большинство из этих компонентов относятся к фреймворку MVC Spring, и у каждого из них есть свои конкретные обязанности.

Пока вы узнали только, что DispatcherServlet — ключевой компонент в обработке запросов с помощью MVC Spring, самое сердце веб-модуля MVC Spring. Именно единая точка входа координирует все действия по обработке запросов аналогично объекту ActionServlet фреймворка Struts и FacesServlet в JSF. Она делегирует обработку запросов инфраструктурным веб-компонентам и обращается к пользовательским веб-компонентам. Она чрезвычайно гибка в настройке и полностью адаптируется к конкретным задачам. Гибкость ей обеспечивает тот факт, что все используемые этим сервлетом компоненты являются интерфейсами для инфраструктурных компонентов. В следующей таблице перечислены некоторые из подобных интерфейсов, предоставляемых фреймворком MVC Spring.

5 Настройка Spring MVC

Основой Spring MVC является сервлет DispatcherServlet , который играет роль входного контроллера в Spring MVC. Как и любой другой сервлет, DispatcherServlet должен быть настроен в файле web.xml веб-приложения. Поэтому первое, что необходимо сделать, – это поместить следующий элемент <servlet> в файл web.xml:

```
      <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/applicationContextMVC.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
       </servlet>
```

Имя сервлета в элементе <servlet-name> имеет большое значение. По умолчанию после загрузки сервлета DispatcherServlet он загрузит контекст приложения Spring из XML-файла с именем, соответствующим имени сервлета. В данном случае, мы изменяем адрес, откуда DispatcherServlet будет брать контекст приложения с помощью изменения параметра `contextConfigLocation` на указанный нами путь `/WEB-INF/applicationContextMVC.xml`.

Для конфигурации в Java-коде необходимо создать класс, который наследуется от AbstractAnnotationConfigDispatcherServletInitializer:

```
      DispatcherServletInitializer                                    extends
AbstractAnnotationConfigDispatcherServletInitializer
```

И необходимо реализовать его методы.

Далее необходимо указать, какие адреса URL будут обрабатываться сервлетом DispatcherServlet. Обычно сервлету DispatcherServlet передаются шаблоны адресов URL, такие как *.htm, /* или /app. Но эти шаблоны имеют следующие проблемы.

- Шаблон *.htm предполагает, что ответ всегда будет иметь формат HTML.
- Шаблон /* не предполагает какого-то определенного формата ответа, но указывает, что DispatcherServlet будет обслуживать все запросы. Это усложняет обслуживание статических объектов, таких как изображения и таблицы стилей.
- Шаблон /app (или подобный ему) помогает отделить содержимое, обслуживаемое сервлетом DispatcherServlet, от остального содержимого. Но тогда нам придется реализовать обработку характерных особенностей наших

URL (в частности, путь /app). Это ведет к сложному преобразованию адресов URL с целью скрыть путь /app.

Вместо того чтобы использовать какие-либо из этих проблематичных схем, я (автор книги Spring in Action) предпочитаю настраивать отображение DispatcherServlet, как показано ниже:

```
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

Отображая сервлет DispatcherServlet в адрес /, мы сообщаем, что он является сервлетом по умолчанию, отвечающим за обработку всех запросов, включая запросы на получение статического содержимого.

Если вас беспокоит, как DispatcherServlet будет обрабатывать эти типы запросов, тогда задержитесь ненадолго. Небольшой трюк с настройками освободит вас как разработчика от необходимости заботиться о деталях. Пространство имен mvc в Spring включает новый элемент <mvc:resources>, автоматически обслуживающий запросы на получение статического содержимого. Все, что от вас требуется, – всего лишь определить его в конфигурации Spring. Это означает, что пришло время создать файл applicationContextMVC.xml, который сервлет DispatcherServlet будет использовать для создания контекста приложения.

```
<mvc:resources mapping="/resources/**" location="/resources/" />
```

Как уже говорилось выше, все запросы, проходящие через Dispat cherServlet, должны каким-то образом обрабатываться, часто с помощью других контроллеров. Поскольку запросы на получение статического содержимого также обрабатываются сервлетом DispatcherServlet, необходимо каким-то образом сообщить ему, как обрабатывать эти ресурсы. Но создание отдельного контроллера для этой цели выглядит слишком трудоемкой задачей. К счастью, эту работу может выполнить элемент <mvc:resources>. Элемент <mvc:resources> определяет обработчик статического содержимого. Атрибуту mapping в примере присвоено значение /resources/**, включающее шаблонный символ в стиле утилиты Ant, который указывает, что путь должен начинаться с /resources и может включать дополнительные элементы пути. Атрибут location определяет местоположение обслуживаемых файлов. Из данной конфигурации следует, что любые запросы, путь в которых начинается с /resources, автоматически будут адресоваться к папке /resources, находящейся в корневом каталоге приложения. То есть все изображения, таблицы стилей, сценарии JavaScript и другие статические ресурсы должны храниться в папке /resources приложения.

Теперь, когда мы разобрались с проблемой обработки статического содержимого, можно подумать о функциональности приложения. Поскольку мы

находимся лишь в начале пути, начнем с создания простой домашней (или главной) страницы.

6 Настройка поддержки аннотаций в Spring MVC

Как упоминалось выше, сервлет DispatcherServlet обращается к одному или нескольким механизмам отображения с целью определить, какому контроллеру передать запрос для обработки. В состав Spring входят несколько реализаций механизмов отображения, включая перечисленные ниже.

- BeanNameUrlHandlerMapping – отображает контроллеры на адреса URL, опираясь на имена компонентов контроллеров.
- ControllerBeanNameHandlerMapping – подобно BeanNameUrl HandlerMapping, отображает контроллеры на адреса URL, также опираясь на имена компонентов контроллеров. Но в данном случае не требуется, чтобы имена компонентов подбирались в соответствии с соглашениями об адресах URL.
- ControllerClassNameHandlerMapping – отображает контроллеры на адреса URL, используя имена классов контроллеров.
- DefaultAnnotationHandlerMapping – отображает запросы на контроллеры методы контроллеров, отмеченные аннотацией @RequestMapping.
- SimpleUrlHandlerMapping – отображает контроллеры на адреса URL, используя свойство-коллекцию, объявленное в контексте приложения Spring.

Использование любого из этих механизмов отображения обычно сводится к его настройке как компонента Spring. Но если требуемый компонент механизма отображения не будет найден, тогда сервлет DispatcherServlet автоматически создаст и будет использовать BeanNameUrlHandlerMapping и DefaultAnnotationHandlerMapping. К счастью, нас в первую очередь интересуют аннотированные классы контроллеров, поэтому реализация DefaultAnnotationHandlerMapping, используемая сервлетом DispatcherServlet по умолчанию, подходит как нельзя лучше.

DefaultAnnotationHandlerMapping отображает запросы на методы контроллеров, отмеченные аннотацией @RequestMapping (с которой мы встретимся в следующем разделе). Но в Spring MVC аннотации могут применяться не только для отображения запросов на методы. В процессе создания контроллеров мы также будем использовать аннотации для связывания параметров запросов с параметрами методов-обработчиков, выполнения проверки и преобразования сообщений. Поэтому одного компонента DefaultAnnotationHandlerMapping будет мало. К счастью, достаточно добавить в файл applicationContextMVC.xml всего одну строку, чтобы обрести возможность использовать все остальные аннотации, имеющиеся в Spring MVC:

```
<mvc:annotation-driven/>
```

В Java-коде подобный функционал реализует аннотация @EnableWebMVC

Создадим контроллер главной страницы.

7 Контроллер главной страницы

Обычно самое первое, что видят посетители веб-сайта, – это главная страница. Это парадный вход, обеспечивающий доступ ко всей функциональности сайта. В случае с нашим приложением, основная задача главной страницы состоит в том, чтобы поприветствовать посетителя. В листинге далее представлен контроллер HomeController – простейший контроллер Spring MVC, обслуживающий запросы на получение главной страницы.

```
@Controller
public class HomeController {
@RequestMapping("/")
public String showHomePage() {
	return "home";
}
}
```

Несмотря на свою простоту реализации контроллера HomeController, о ней многое можно сказать. Во-первых, аннотация @Controller указывает, что этот класс является классом контроллера. Данная аннотация является специализированной версией аннотации @Component, которая помогает элементу <context:component-scan> отыскивать и регистрировать классы с аннотацией @Controller как компоненты, как если бы они были отмечены аннотацией @Component. Это означает, что в файле applicationContextMVC.xml необходимо настроить элемент <context:component-scan> , чтобы обеспечить автоматическое обнаружение и регистрацию класса HomeController (и всех других классов контроллеров, которые нам предстоит написать) как компонента. Ниже приводится соответствующий фрагмент в XML-файле:

```
<context:component-scan base-package="ru.testmvcapp"/>
```

Аннотация @RequestMapping предназначена для того, чтобы задать методам вашего контроллера адреса, по которым они будут доступны на клиенте.

8 Поиск представлений

Последнее , что осталось сделать в ходе обработки запроса – отобразить страницу. Для решения подобных задач используются представления – обычно JavaServer Pages (JSP), но могут использоваться и другие технологии реализации представлений, такие как Velocity и FreeMarker. Чтобы определить,

какое представление должно обрабатывать данный запрос, DispatcherServlet обращается за помощью к арбитру представлений с целью заменить логическое имя, возвращаемое контроллером, ссылкой на фактическое представление, реализующее отображение результатов.

В действительности работа арбитра представлений заключается в отображении логического имени на некоторую реализацию интерфейса org.springframework.web.servlet.View. Но пока достаточно представлять себе арбитр представлений как некоторый механизм отображения имени представления в JSP, что он и делает. В состав Spring входят несколько реализаций арбитров представлений, таких как BeanNameViewResolver, ContentNegotiatingViewResolver, FreeMarkerViewResolver, InternalResourceViewResolver, JasperReportsViewResolver, ResourceBundleViewResolver, TilesViewResolver, UrlBasedViewResolver, VelocityLayoutViewResolver, VelocityViewResolver, XmlViewResolver.

О них вы можете почитать самостоятельно. У нас недостаточно времени, чтобы познакомиться с каждым из этих арбитров представлений. Но некоторые из них стоят того, чтобы взглянуть на них поближе. Начнем с арбитра InternalResolverViewResolver.

8.1 Поиск внутренних представлений

Фреймворк Spring MVC во многом следует принципу *преимущества соглашений перед настройками*. Арбитр InternalResourceViewResolver является одним из таких элементов, ориентированных на соглашения. Он отображает логическое имя представления в объект View, который перекладывает ответственность за отображение страницы на шаблон (обычно JSP), находящийся в контексте веб-приложения. Как показано на рис. 5, он выполняет такое отображение, добавляя приставку и окончание к логическому имени представления, в результате получая путь к шаблону, являющемуся ресурсом внутри веб-приложения.

Рисунок 5 — InternalResourceViewResolver определяет путь к шаблону представления, добавляя указанные приставку и окончание к логическому имени представления

Допустим, что все страницы JSP для нашего приложения находятся в каталоге /WEB-INF/views/. Учитывая это условие, компонент InternalResourceViewResolver необходимо настроить в файле конфигурации, как показано ниже:

```
<bean id="templateResolver" class="org.springframework.web.servlet
.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/views/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

Когда DispatcherServlet запросит у арбитра InternalResourceViewResolver представление, он добавит к логическому имени приставку /WEB-INF/views/ и окончание .jsp. В результате получится путь к странице JSP для отображения вывода. Затем InternalResourceViewResolver передаст этот путь объекту View, который направит запрос странице JSP. То есть, когда HomeController вернет логическое имя представления home, в конечном итоге будет получен путь /WEB-INF/views/home.jsp.

По умолчанию объект View, созданный арбитром InternalResourceView Resolver, является экземпляром класса InternalResourceView, который просто передает запрос странице JSP для отображения.

8.2 FreeMarker

Далее в качестве примеров будет использоваться шаблонизатор FreeMarker.

FreeMarker — это шаблонизатор на основе Java от Apache Software Foundation. Как и другие движки шаблонов, FreeMarker предназначен для поддержки веб-страниц HTML в приложениях, следующих шаблону MVC. FreeMarker как и многие другие шаблонизаторы используются в качестве в качестве альтернативы JSP.

Настройка FreeMarker выглядит следующим образом:

```
<bean id="freeMarkerConfig" class="org.springframework.web.servlet.view.
freemarker.FreeMarkerConfigurer">
	<property name="templateLoaderPath" value="/WEB-INF/views"/>
	<property name="defaultEncoding" value="UTF-8"/>
	<property name="freemarkerSettings">
		<props>
			<prop key="default_encoding">UTF-8</prop>
		</props>
	</property>
</bean>

<bean id="templateResolver" class="org.springframework.web.servlet.view.
	reemarker.FreeMarkerViewResolver">
	<property name="suffix" value=".ftl"/>
	<property name="contentType" value="text/html;charset=UTF-8"/>
	<property name="cache" value="false"/>
</bean>
```

Кратко, FreeMarker работает следующим образом:

На вход подается шаблон, например html в котором есть специальные выражения, подготавливаются данные соответствующие этим выражением, а Freemarker динамически вставляет эти данные и получается динамически заполненный документ.

К примеру отобразим простейшее приветственное сообщение с помощью FreeMarker. Первым делом создадим шаблон .ftl.

```
<meta charset="UTF-8">
<html>
<head>
	<title>Здравствуйте!</title>
</head>

<body>
	<h1>Вот ваше сообщение:</h1>
	<h2>${message}</h2>
</body>
</html>
```

Контроллер будет выглядеть следующим образом:

```
@Controller
public class HelloController {
@RequestMapping({"/hello"})
public String showHelloPage(Model model) {
	model.addAttribute("message", "Это ваше сообщение!");
	return "hello";
}
}
```

9 Контроллер обработки входных данных

Контроллер HelloController получился довольно простым. Перед ним не стоит задача обрабатывать пользовательские данные или какие-либо параметры. Он просто выводит приветственное сообщение. Сложно было бы придумать более простой контроллер.

Но не у всех контроллеров жизнь такая же простая. Часто от контроллеров требуется выполнять операции с некоторыми данными, которые передаются в виде параметров запроса в URL или данных формы.

Приведем также простой пример, допустим мы хотим отобразить имя, полученное в URL запроса. Приступим к реализации данной функциональности и посмотрим, как писать контроллеры, обрабатывающие входные данные.

9.1 Создание контроллера, обрабатывающего входные данные

```
@Controller
@RequestMapping({"/hello"})
public class HelloController {
	@RequestMapping(value="/", method=RequestMethod.GET)
	public String showHelloPage(@RequestParam(value = "name",
	required = false, defaultValue = "незнакомец") String name, Model model) {
	model.addAttribute("message", "Привет, " + name + "!");
	return "hello";
}
}
```

```
<meta charset="UTF-8">
<html>
<head>
	<title>Здравствуйте!</title>
```

```
</head>

<body>
	<h2>${message}</h2>
</body>
</html>
```

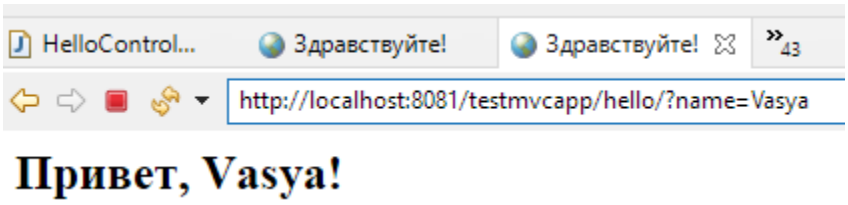

Рисунок 6 — Пример простой обработки входных данных

В листинге выше класс HomeController отмечен аннотациями @Controller и @RequestMapping . Как уже говорилось выше, аннотация @Controller подсказывает элементу <context:component-scan>, что данный класс должен автоматически регистрироваться в контексте приложения Spring как компонент.

Обратите внимание, что класс HelloController также отмечен аннотацией @RequestMapping. Аннотация @RequestMapping уже использовалась в определении класса HomeController для аннотирования метода showHomePage(), но при применении на уровне класса аннотация @RequestMapping действует несколько иначе.

При применении к классу, как в данном примере, аннотация @RequestMapping определяет корневой путь в URL, обрабатываемый данным контроллером. В данном случае аннотация @RequestMapping сообщает, что все запросы будут иметь пути, начинающиеся с /hello. В настоящий момент в классе HelloController имеется всего один метод: showHelloPage(). Подобно любому другому методу-обработчику, данный метод отмечен аннотацией @RequestMapping. Она мало чем отличается от аналогичной аннотации, использованной в классе HomeController. Но в действительности разница гораздо существеннее, чем кажется на первый взгляд.

Аннотация @RequestMappings ограничивает область отображения, объявленную в аннотации @RequestMapping на уровне класса. В данном случае на уровне класса класс HelloController отображается на путь /hello, а на уровне метода – на путь «/». В результате объединения этих двух аннотаций получается, что метод showHelloPage() должен обрабатывать запросы для /hello/. Кроме того, значение RequestMethod.GET в атрибуте method указывает, что этот метод будет обрабатывать только запросы типа HTTP GET.

Метод showHelloPage() принимает строковый параметр name с именем пользователя и объект Model.

Параметр name отмечен аннотацией @RequestParam("name"), чтобы показать, что он должен получить значение параметра запроса Этот параметр будет использоваться методом showHelloPage() для вывода сообщения. В

Вторым передается объект Model. В действительности объект, передаваемый как параметр типа Model, можно передавать как параметр типа Map<String,Object>. Но тип Model позволяет использовать некоторые вспомогательные методы, удобные для заполнения модели, такие как addAttribute(). Метод addAttribute() действует практически так же, как метод put() интерфейса Map, за исключением того, что он определяет собственный ключ для доступа к данным в отображении.

9.2 Обработка форм

При работе с формами в веб-приложении выполняются две операции: отображение формы и обработка данных, отправленных пользователем вместе с формой. Таким образом, продемонстрируем две эти операции. Прежде чем форма с данными попадет на сервер, она должна быть отображена в браузере, поэтому начнем с метода-обработчика, реализующего отображение формы ввода информации о студенте (из нашей предыдущей лекции).

Для отображения формы необходим объект Student, свойства которого будут связаны с полями формы. Поскольку форма предназначена для ввода данных о студенте, достаточно будет создать новый, неинициализированный объект Student. В листинге ниже представлен метод-обработчик createStudent(), создающий объект Student и помещающий его в модель.

```
@Controller
@RequestMapping({"/add"})
public class AddStudentController {
@RequestMapping(method=RequestMethod.GET, params="new")
public String createStudent(Model model) {
	model.addAttribute(new StudentForm());
	return "studentform";
	}
```

Подобно другим методам-обработчикам, createStudent() отмечен аннотацией @RequestMapping . Данный метод обрабатывает запросы для пути в URL, определяемом в аннотации @RequestMapping на уровне класса, в данном случае /add. Единственное, что определяет аннотация @RequestMapping для этого метода, что он будет обрабатывать только запросы типа HTTP GET. Обратите также внимание на атрибут params , которому присваивается значение new. Это означает, что данный метод будет обрабатывать запросы HTTP GET для пути /add в URL, только если запрос включает параметр new.

Атрибут params аннотации @RequestMapping может ограничивать область применения метода-обработчика запросами, содержащими определенные параметры Для внутренних нужд метод createStudent() создает новый экземпляр Student и добавляет его в модель. По завершении он возвращает логическое имя представления studentForm, реализующего отображение формы. Теперь создадим само представление.

9.2.1 Определение представления формы

```
    <#import "/spring.ftl" as spring/>
    <meta charset="UTF-8">
    <html>
    <head>
   <title>Добавление нового студента</title>
</head>
<body>
   <div>
      <fieldset>
         <legend>Добавление студента</legend>
         <form name="student" action="" method="POST">
          <table cellspacing="0">
           <tr>
            <th><label>Фамилия:</label><th>
            <td><@spring.formInput "studentForm.lastName" "" "text"/><br>
            <@spring.showErrors "<br>" /></td>
           </tr>
           <tr>
            <th><label>Имя:</label><th>
            <td><@spring.formInput "studentForm.firstName" "" "text"/><br>
            <@spring.showErrors "<br>" /></td>
          </tr>
          <tr>
          <th><label>Отчество:</label><th>
          <td><@spring.formInput "studentForm.patronymic" "" "text"/><br>
          <@spring.showErrors "<br>" />
          </td>
          </tr>
          <tr><th><label>Средний балл:</label><th>
          <td><@spring.formInput "studentForm.avgMark" "" "text"/><br>
          <@spring.showErrors "<br>" />
          </td>
          </tr>
          </table>
            <input type="submit" value="Добавить" />
         </form>
      </fieldset>
   </div>
</body>
    </html>
```

9.3 Обработка данных формы

Реализовав возможность отправки формы пользователем, мы должны создать метод-обработчик, принимающий объект Student (заполненный данными из формы) и выводящий сообщение об успешном добавлении.

```
@RequestMapping(method=RequestMethod.POST)
public String addStudentFromForm(@Valid StudentForm form,
BindingResult bindingResult) {
      if(bindingResult.hasErrors()) {
            return "studentform";
      }
// здесь могло быть сохранение объекта в бд
return "redirect:add/" + form.getFirstName(); // Переадресовать
}

@RequestMapping(value="/{name}", method=RequestMethod.GET)
```

```
public String helloStudent(@PathVariable String name, Model model) {
	model.addAttribute("message", "Привет, " + name + "!");
	return "hello";
	}
```

Обратите внимание, что метод addStuidentFromForm() отмечен аннотацией @RequestMapping, мало отличающейся от предыдущего примера. Ни одна из них не определяет путь в URL, поэтому оба метода будут обрабатывать запросы для пути /add в URL. Разница состоит в том, что createStudent() обрабатывает GET-запросы, а addSStudentFromForm() обрабатывает POST-запросы. Здесь все в порядке, потому что именно так отправляются формы. После отправки формы ее поля в запросе будут связаны со свойствами объекта StudentForm, который затем будет передан методу addStudentFromForm().

В листинге выше можно также заметить, что параметр StudentForm отмечен аннотацией @Valid. Она указывает, что объект StudentForm должен подвергаться проверке перед передачей методу. Подробнее об этой проверке рассказывается в следующем разделе. Подобно методам-обработчикам, созданным ранее, этот метод также возвращает строку, определяющую, куда дальше должен направляться запрос. На этот раз вместо логического имени представления возвращается специальное представление переадресации. Префикс redirect: свидетельствует о том, что запрос должен быть переадресован в путь, указанный вслед за префиксом. Выполняя переадресацию на другую страницу, можно избежать двойной отправки формы, если пользователь щелкнет на кнопке Refresh (Обновить) в браузере. Что касается пути переадресации – он примет вид /add/{name}, где {name} представляет только что полученное имя студента.

9.3.1 Обработка запросов, в которых путь определяется переменной

Теперь необходимо разобраться, как реализовать обработку запросов к пути /add/{name}. Для этого нужно добавить в AddStudentController еще один метод-обработчик:

Метод helloStudent() не слишком отличается от других методов-обработчиков, созданных до сих пор. Он принимает строковый параметр с именем студента и отображает информацию с помощью уже известного вам представления hello.

Однако метод helloStudent() имеет ряд отличительных особенностей. Во-первых, атрибут value в аннотации @RequestMapping содержит странные фигурные скобки. А во-вторых, параметр name отмечен аннотацией @PathVariable . Вместе эти две особенности позволяют методу helloStudent() обрабатывать запросы к адресам URL с параметрами в их путях. Фрагмент пути {name} фактически является меткой-заполнителем, соответствующей параметру name метода, отмеченному аннотацией @PathVariable. Независимо от

того, что находится в соответствующем фрагменте пути, эта часть строки будет передана методу в виде параметра name.

9.4 Проверка входных данных

К процедуре создания нового пользователя предъявляются определенные требования. В частности, нам необходимо чтобы были введены полное ФИО и корректный средний балл.

Аннотация @Valid – первая линия обороны от недопустимых входных данных. В действительности аннотация @Valid является частью спецификации «JavaBean Validation API». Версия Spring 3 включает поддержку JSR-303, и мы воспользовались аннотацией @Valid, чтобы сообщить фреймворку Spring, что объект StudemtForm должен быть проверен, так как значения своих свойств он получает из данных, введенных пользователем.

Если в процессе проверки объекта StudentForm будут обнаружены недопустимые данные, информация об ошибках будет передана методу addStudentFromForm() в виде объекта BindingResult во втором параметре. Если метод BindingResult.hasErrors() вернет true, это означает, что проверка потерпела неудачу. В этом случае метод вернет логическое имя представления studentform, чтобы обеспечить повторное отображение формы и дать пользователю возможность исправить ошибки.

Но как Spring отличит допустимый объект Student от недопустимого?

Кроме всего прочего, спецификация JSR-303 определяет несколько аннотаций, определяющих правила проверки допустимости значений в свойствах. Чтобы определить «допустимость» каждого свойства объекта StudentForm, можно использовать эти аннотации. В листинге ниже представлены свойства класса StudentForm отмеченные аннотациями с правилами проверки.

```
@NotEmpty(message="Поле не должно быть пустым")
private String firstName;

@NotEmpty(message="Поле не должно быть пустым")
private String lastName;

@NotEmpty(message="Поле не должно быть пустым")
private String patronymic;

@NotEmpty(message="Поле не должно быть пустым")
@Pattern(regexp="[2-5]\\.?[0-9]*", message="Проверье правильность ввода числа")
private String avgMark;
```

Первые три свойства отмечены аннотацией @NotEmpty, определяемой спецификацией JSR-303, реализующей проверку отсутствия значения аннотированного свойства.

Чтобы убедиться, что значение свойства avgMark формально соответствует формату числа в диапазоне от 2 до 5, оно было отмечено

аннотацией @Pattern с регулярным выражением в атрибуте regexp, на соответствие которому должно проверяться значение свойства

Во всех аннотациях выше, реализующих проверку, определяется атрибут message с текстом сообщения, отображаемым в форме, в случае, когда проверка терпит неудачу, чтобы пользователь знал, какое значение следует исправить. Теперь, когда пользователь отправит форму регистрации методу addStudentFromForm() контроллера AddStudentController, значения полей объекта StudentForm будут проверены в соответствии с правилами в аннотациях. Если какое-либо из правил будет нарушено, метод-обработчик вновь отправит пользователя к форме ввода, чтобы он мог исправить ошибки. При повторной отправке формы пользователю необходимо иметь некоторый механизм, который позволил бы описать проблему. Поэтому вернемся к JSP-файлу с определением формы и добавим код, отображающий сообщения механизма проверки.

Объект BindingResult, переданный методу addStudentFromForm(), позволяет определить, были ли выявлены ошибки в процессе проверки формы. Для этого достаточно просто вызвать его метод hasErrors(). Но в этом объекте также содержатся сообщения об ошибках для полей, не прошедших проверку. Один из способов отобразить текст сообщений об ошибках состоит в том, чтобы извлекать сообщения об ошибках для отдельных полей вызовом метода getFieldError() объекта BindingResult. Но в FreeMarker имеется более удобный способ – с использованием макроса @spring.showErrors. (как показано в 9.2.1)

10* Некоторые примечания:

Для отображения ссылок в FreeMarker можно воспользоваться тегом <a> и макросом `@spring.url`

```
<a href="<@spring.url'/add?new'/>">Добавь нового!</a>
```

Для отображения списка можно воспользоваться макросом #list, подробнее:

https://freemarker.apache.org/docs/ref_directive_list.html