

Spring Web MVC



Григорий
Вахмистров



Григорий Вахмистров

Backend Developer в Tennisi.bet

План занятия

1. [Предисловие](#)
2. [Spring Web MVC](#)
3. [HttpMessageConverters](#)
4. [Итоги](#)
5. [Домашнее задание](#)



Предисловие



Предисловие

На прошлой лекции мы посмотрели на **основы работы Spring** и возможности **конфигурирования с помощью XML, аннотаций и Java**.

Сегодня наша задача - посмотреть, каким образом это встраивается в разработку веб-приложений и как создать своё первое REST API.



Spring Web MVC



Spring Web MVC

Spring Web MVC (иногда просто Spring Web или Spring MVC)
- web framework, написанный поверх Servlet API (именно
поэтому мы и изучали сервлеты).



Embed Tomcat

На этот раз мы обойдёмся без полноценного Tomcat, а возьмём встраиваемую версию.

Это поможет проще разрабатывать и отлаживать приложение.


```
<dependencies>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-core</artifactId>
    <version>9.0.39</version>
  </dependency>
  <dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <version>9.0.39</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.9.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.6</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

Embed Tomcat

```
public class Main {  
    public static void main(String[] args) throws LifecycleException, IOException {  
        final var tomcat = new Tomcat();  
        final var baseDir :Path = Files.createTempDirectory( prefix: "tomcat");  
        baseDir.toFile().deleteOnExit();  
        tomcat.setBaseDir(baseDir.toAbsolutePath().toString());  
  
        final var connector = new Connector();  
        connector.setPort(9999);  
        tomcat.setConnector(connector);  
  
        tomcat.getHost().setAppBase(".");  
        tomcat.addWebapp( contextPath: "", docBase: ".");  
  
        tomcat.start();  
        tomcat.getServer().await();  
    }  
}
```

По большей части это инфраструктурный код, не имеющий никакого отношения к Spring и необходимый только для запуска Tomcat.



DispatcherServlet

Первое и ключевое, что нам предоставляет Spring Web MVC - это `DispatcherServlet`. Это значительно улучшенный аналог нашего `MainServlet`'а, который и занимается диспетчеризацией запросов.

Если бы мы работали с `web.xml`, мы спокойно зарегистрировали его там. Но мы не работаем с `web.xml`.



DispatcherServlet

Вы можете прочитать спецификацию сервлетов и увидеть, что сервлеты можно регистрировать с помощью аннотаций `@WebServlet`

Т.е. нам придётся отнаследоваться от DispatcherServlet'a и поставить над своим классом аннотацию.

ServletContainerInitializer

Но есть другой способ, заключающийся в следующем:

на самом деле Tomcat ищет все классы, имплементирующие интерфейс:

```
public interface ServletContainerInitializer {  
  
    /** Notifies this <tt>ServletContainerInitializer</tt> of the startup ...*/  
    public void onStartUp(Set<Class<?>> c, ServletContext ctx)  
        throws ServletException;  
}
```

И запускает на них метод `onStartup`.



SpringServletContainerInitializer

Spring уже предоставляет готовую реализацию этого интерфейса, которая ищет уже классы, имплементирующие `WebApplicationInitializer` и запускает на них метод `onStartup`.

Это вообще отличительная особенность Spring'a: написание удобных и универсальных адаптеров к различным инструментам и стандартам.

WebApplicationInitializer

```
public class ApplicationInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext servletContext) {
        final var context = new AnnotationConfigWebApplicationContext();
        context.scan( ...basePackages: "ru.netology");
        context.refresh();

        final var servlet = new DispatcherServlet(context);
        final var registration : ServletRegistration.Dynamic = servletContext.addServlet(
            servletName: "app", servlet
        );
        registration.setLoadOnStartup(1);
        registration.addMapping( ...urlPatterns: "/" );
    }
}
```

Для веб-приложений предоставляются собственные контексты, которые автоматически не рефрешатся.



@RequestMapping

Помимо вышеобозначенного, Spring Web предоставляет аннотацию `@RequestMapping`, позволяющую привязать Controller и его методы к определённым путям (см.следующий слайд).

@RequestMapping

```
@Controller
@RequestMapping("/api/posts")
public class PostController {
    public static final String APPLICATION_JSON = "application/json";
    private final PostService service;

    public PostController(PostService service) { this.service = service; }

    @GetMapping
    public void all(HttpServletResponse response) throws IOException {...}

    @GetMapping("/{id}")
    public void getById(@PathVariable long id, HttpServletResponse response) {}

    @PostMapping
    public void save(HttpServletRequest request, HttpServletResponse response) throws IOException {...}

    @DeleteMapping("/{id}")
    public void removeById(long id, HttpServletResponse response) {}
}
```

@RequestMapping

@GetMapping, @PostMapping и т.д., просто convenience-аннотации для выбора по соответствующему HTTP-методу:

```
@RequestMapping(method = RequestMethod.GET)  
public @interface GetMapping {
```

@RequestMapping

Mapping'и строятся иерархически, начиная от класса.

Т.е. у handler'а `all` будет путь `"/api/posts"`, а у `getById` - `"/api/posts/{id}"`.

Под handler'ами мы будем понимать все методы, над которыми стоит аннотация `@RequestMapping`.

@PathVariable

RequestMapping'и поддерживает концепции плейсхолдеров для переменных пути в "{id}" (полный путь "/api/posts/{id}") всё, что идёт после "/api/posts" до следующего слеша складывается в переменную id, которая затем подставляется в параметр id, если перед ним стоит аннотация @PathVariable и имя параметра совпадает с именем плейсхолдера:

```
@GetMapping("/{id}")  
public void getById(@PathVariable long id, HttpServletResponse response) {}
```



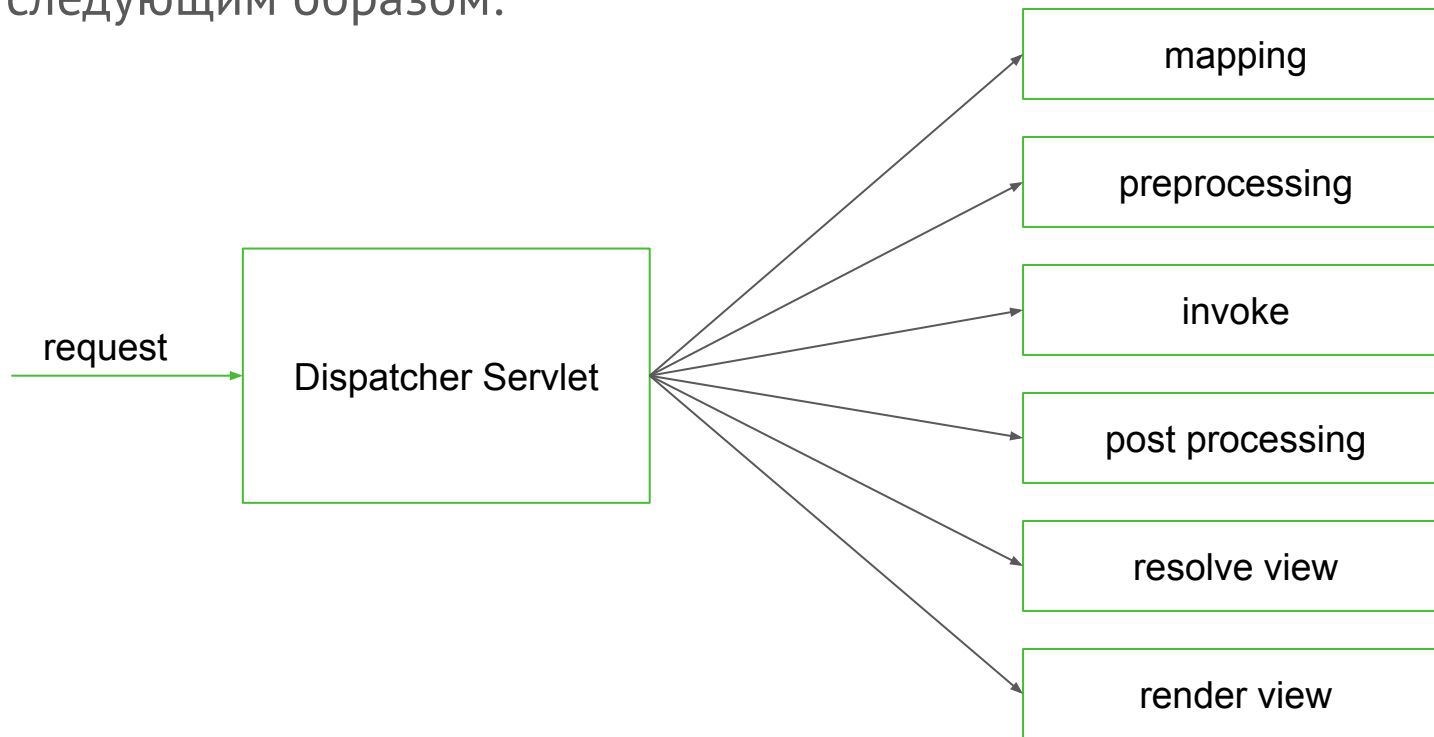
HandlerAdapter'ы

За подготовку вызова соответствующего метода отвечают
HandlerAdapter'ы:

именно они подготавливают вызов метода, предоставляя нужные
параметры (например, вытаскивая их из PathVariable или
передавая HttpServletRequest/Response).

Классические приложения

Общая модель для классических приложений выглядит следующим образом:



MVC

Эта модель была разработана под паттерн MVC:

- Model
- View
- Controller

View чаще всего собой представляла HTML-страницу, Excel-файл, PDF и т.д.

REST

Для современной модели (особенно, если мы хотим делать REST-сервисы), она не особо подходит, т.к. HTML у нас как такового нет, есть JSON. И хотелось бы, чтобы то, что мы делаем сейчас руками:

```
@GetMapping
public void all(HttpServletResponse response) throws IOException {
    response.setContentType(APPLICATION_JSON);
    final var data : List<Post> = service.all();
    final var gson = new Gson();
    response.getWriter().print(gson.toJson(data));
}
```

происходило автоматически, т.к. никакой логики тут нет, обычная конвертация в JSON.

REST

Если же мы попробуем преобразовать в:

```
@GetMapping  
public List<Post> all() {  
    return service.all();  
}
```

Ничего не получится, потому что Spring не понимает, как преобразовать `List<Post>` в имя View.

@ResponseBody, @RequestBody

Spring предоставляет аннотации `@ResponseBody` и `@RequestBody`, которые “пытаются” трансформировать ответ из метода - в тело ответа и тело запроса - в параметр соответственно. (На самом деле это делают не сами аннотации, а соответствующие классы, к которым попадает информация о наличии данных аннотаций).

Стоит отметить, что если мы делаем REST Service, то логично над всеми handler'ами поставить `@ResponseBody`. Для этого достаточно `@Controller` заменить на `@RestController` и это произойдёт автоматически.

@ResponseBody, @RequestBody

```
@RestController
@RequestMapping("/api/posts")
public class PostController {
    public static final String APPLICATION_JSON = "application/json";
    private final PostService service;

    public PostController(PostService service) { this.service = service; }

    @GetMapping
    public List<Post> all() {
        return service.all();
    }
}
```



@ResponseBody, @RequestBody

Но теперь мы получим следующий Exception:

```
No converter found for return value of type: class java.util.Collections$EmptyList
```

Spring по-прежнему не знает, как конвертировать [List](#) в тело ответа.



HttpMessageConverters

HttpMessageConverter

```
/** Strategy interface that specifies a converter that can convert from and to HTTP requests and responses. ...*/
public interface HttpMessageConverter<T> {

    /** Indicates whether the given class can be read by this converter. ...*/
    boolean canRead(Class<?> clazz, @Nullable MediaType mediaType);

    /** Indicates whether the given class can be written by this converter. ...*/
    boolean canWrite(Class<?> clazz, @Nullable MediaType mediaType);

    /** Return the list of {@link MediaType} objects supported by this converter. ...*/
    List<MediaType> getSupportedMediaTypes();

    /** Read an object of the given type from the given input message, and returns it. ...*/
    T read(Class<? extends T> clazz, HttpInputMessage inputMessage)
        throws IOException, HttpMessageNotReadableException;

    /** Write an given object to the given output message. ...*/
    void write(T t, @Nullable MediaType contentType, HttpOutputMessage outputMessage)
        throws IOException, HttpMessageNotWritableException;

}
```

RequestMappingHandlerAdapter

Т.е. можно написать свой Converter. Но Spring как всегда уже адаптировал самые популярные решения и для Gson у нас уже есть [GsonHttpMessageConverter](#). Нужно лишь зарегистрировать в [RequestMappingHandlerAdapter](#)'е:

```
@Configuration
public class WebConfig {
    @Bean
    public RequestMappingHandlerAdapter requestMappingHandlerAdapter() {
        final var bean = new RequestMappingHandlerAdapter();
        bean.getMessageConverters().add(new GsonHttpMessageConverter());
        return bean;
    }
}
```

RequestMappingHandlerAdapter

Уже содержит реализацию базовых конвертеров:

```
public RequestMappingHandlerAdapter() {  
    this.messageConverters = new ArrayList<>(initialCapacity: 4);  
    this.messageConverters.add(new ByteArrayHttpMessageConverter());  
    this.messageConverters.add(new StringHttpMessageConverter());  
    try {  
        this.messageConverters.add(new SourceHttpMessageConverter<>());  
    }  
    catch (Error err) {  
        // Ignore when no TransformerFactory implementation is available  
    }  
    this.messageConverters.add(new AllEncompassingFormHttpMessageConverter());  
}
```

И мы просто добавляем свой конвертер.



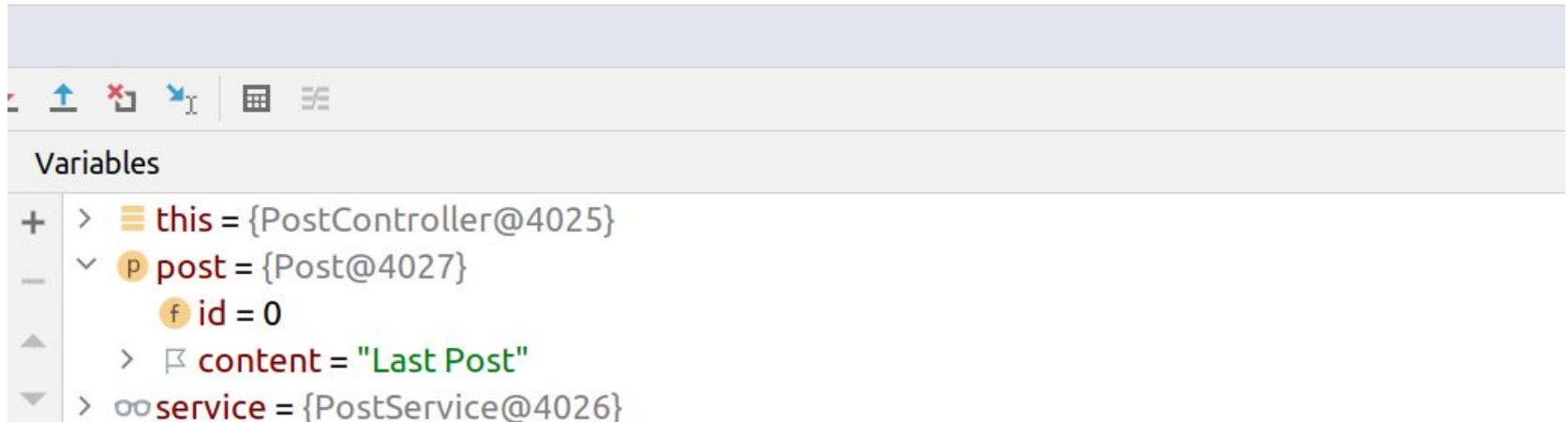
RequestMappingHandlerAdapter

Это позволяет нам читать сразу из тела запроса и писать в тело ответа (снимая необходимость во View).

RequestMappingHandlerAdapter

Это позволяет нам считывать и тело:

```
@PostMapping
public Post save(@RequestBody Post post) {
    return service.save(post);
}
```



The screenshot shows the 'Variables' window in an IDE. It displays the state of variables during a POST request. The variables are:

- `this` = {PostController@4025}
- `post` = {Post@4027}
 - `id` = 0
 - `content` = "Last Post"
- `service` = {PostService@4026}

Итоговый код

```
@RestController
@RequestMapping("/api/posts")
public class PostController {
    private final PostService service;

    public PostController(PostService service) { this.service = service; }

    @GetMapping
    public List<Post> all() {
        return service.all();
    }

    @GetMapping("/{id}")
    public Post getById(@PathVariable long id) {
        return service.getById(id);
    }

    @PostMapping
    public Post save(@RequestBody Post post) {
        return service.save(post);
    }

    @DeleteMapping("/{id}")
    public void removeById(long id) {
        service.removeById(id);
    }
}
```

Как вы видите, код стал гораздо чище: мы избавились от кучи инфраструктурных вызовов и фактически, занимаемся только логикой приложения.



Tuning

То, как мы настраивали Spring - это достаточно редкое явление, используемое тогда, когда вы хотите выжать из него максимум по производительности при минимуме расхода ресурсов (при этом требуется достаточно хороший уровень понимания того, как устроен Spring и Spring Web MVC внутри).

Хотя иногда на собеседованиях спрашивают о навыках настройки Spring с нуля (т.к. большинство кандидатов не понимает, как внутри всё работает).

Spring Boot

В большинстве же приложений стараются идти по другому пути, поскольку понятно, что ряд типовых возможностей уже можно реализовать “из коробки”:

- инициализацию встроенного Tomcat'a;
- создание контекста приложения и регистрацию DispatcherServlet'a;
- регистрацию базовых MessageConverter'ов на основании того, что уже есть в ClassPath (например, если там есть Gson, то можно сразу автоматически регистрировать конвертер на базе него).

Spring Boot

За это и отвечает Spring Boot, настраивающий за вас большую часть по умолчанию. С ним вы и познакомитесь на следующей лекции. И вся наша "тонкая настройка" выльется в:

```
@SpringBootApplication
public class RestApplication {

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

}
```



Итоги



Итоги

Сегодня мы посмотрели на основы использования Spring WebMVC и ключевые аннотации. На следующих лекциях мы будем использовать автоконфигурацию, предоставляемую Spring Boot и детально разберём все используемые аннотации и типовые задачи вроде валидации и обработки исключений.



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера .
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

⌘ нетология

**Задавайте вопросы и
пишите отзыв о лекции!**

Григорий Вахмистров