(/)

Build a REST API with Spring and Java Config

Last modified: January 25, 2020

by Eugen Paraschiv (https://www.baeldung.com/author/eugen/)

X

1. Overview

X

This article shows how to **set up REST in Spring** – the Controller and HTTP response codes, configuration of payload marshalling and content negotiation.

Further reading:

Using Spring @ResponseStatus to Set HTTP Status Code (https://www.baeldung.com/spring-response-status)

Have a look at the @ResponseStatus annotation and how to use it to set the response status code.

Read more (https://www.baeldung.com/spring-response-status) →

The Spring @Controller and @RestController Annotations (https://www.baeldung.com/spring-controller-vs-restcontroller)

X

2. Understanding REST in Spring

The Spring framework supports two ways of creating RESTful services:

- using MVC with *ModelAndView*

team understood this and provided first-class REST support starting with Spring 3.0.

The new approach, based on *HttpMessageConverter* and annotations, is much more lightweight and easy to implement. Configuration is minimal, and it provides sensible defaults for what you would expect from a RESTful service.

3. The Java Configuration

X

```java
@Configuration
@EnableWebMvc
public class WebConfig{
   //
}
```

The new *@EnableWebMvc* annotation does some useful things – specifically, in the case of REST, it detects the existence of Jackson and JAXB 2 on the classpath and automatically creates and registers default JSON and XML converters. The functionality of the annotation is equivalent to the XML version:

<mvc:annotation-driven />

This is a shortcut, and though it may be useful in many situations, it's not perfect. When more complex configuration is needed, remove the annotation and extend *WebMvcConfigurationSupport* directly.

3.1. Using Spring Boot

X

applications.html#boot-features-spring-mvc-auto-configuration).

We can still add MVC functionality to this configuration by implementing the *WebMvcConfigurer* interface on a *@Configuration* annotated class. We can also use a *WebMvcRegistrationsAdapter* instance to provide our own *RequestMappingHandlerMapping*, *RequestMappingHandlerAdapter*, or *ExceptionHandlerExceptionResolver* implementations.

Finally, if we want to discard Spring Boot's MVC features and declare a custom configuration, we can do so by using the *@EnableWebMvc* annotation.

4. Testing the Spring Context

Starting with Spring 3.1, we get first-class testing support for *@Configuration* classes:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
  classes = {WebConfig.class, PersistenceConfig.class},
  loader = AnnotationConfigContextLoader.class)
public class SpringContextIntegrationTest {

   @Test
   public void contextLoads(){
      // When
   }
}
```

Notice that the *WebConfig* configuration class was not included in the test because it needs to run in a Servlet context, which is not provided.

4.1. Using Spring Boot

Spring Boot provides several annotations to set up the Spring *ApplicationContext* for our tests in a more intuitive way.

X

We can load only a particular slice of the application configuration, or we can simulate the whole context startup process.

For instance, we can use the *@SpringBootTest* annotation if we want the entire context to be created without starting the server.

With that in place, we can then add the *@AutoConfigureMockMvc* to inject a *MockMvc* instance and send HTTP requests:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class FooControllerAppIntegrationTest {
```

X

```
    public void whenTestApp_thenEmptyResponse() throws Exception {
        this.mockMvc.perform(get("/foos")
            .andExpect(status().isOk())
            .andExpect(...);
    }

}
```

To avoid creating the whole context and test only our MVC Controllers, we can use *@WebMvcTest:*

```java
@RunWith(SpringRunner.class)
@WebMvcTest(FooController.class)
public class FooControllerWebLayerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private IFooService service;

    @Test()
    public void whenTestMvcController_thenRetrieveExpectedResult() throws Exception {
        // ...

        this.mockMvc.perform(get("/foos")
            .andExpect(...);
    }
}
```

We can find detailed information on this subject on our 'Testing in Spring Boot' article

5. The Controller

The *@RestController* is the central artifact in the entire Web Tier of the RESTful API. For the purpose of this post, the controller is modeling a simple REST resource – *Foo*:

```java
@RestController
```

```
@RequestMapping("/foos")
class FooController {

    @Autowired
    private IFooService service;

    @GetMapping
    public List<Foo> findAll() {
        return service.findAll();
    }

    @GetMapping(value = "/{id}")
    public Foo findById(@PathVariable("id") Long id) {
        return RestPreconditions.checkFound(service.findById(id));
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Long create(@RequestBody Foo resource) {
        Preconditions.checkNotNull(resource);
        return service.create(resource);
    }
```

X

X

```
        RestPreconditions.checkNotNull(service.getById(resource.getId()));
        service.update(resource);
    }

    @DeleteMapping(value = "/{id}")
    @ResponseStatus(HttpStatus.OK)
    public void delete(@PathVariable("id") Long id) {
        service.deleteById(id);
    }

}
```

You may have noticed I'm using a straightforward, Guava style *RestPreconditions* utility:

```
public class RestPreconditions {
    public static <T> T checkFound(T resource) {
        if (resource == null) {
            throw new MyResourceNotFoundException();
        }
        return resource;
    }
}
```

The Controller implementation is non-public – this is because it doesn't need to be.

Usually, the controller is the last in the chain of dependencies. It receives HTTP requests from the Spring front controller (the *DispatcherServlet*) and simply delegates them forward to a service layer. If there's no use case where the controller has to be injected or manipulated through a direct reference, then I prefer not to declare it as public.

return type.

The *@RestController* is a shorthand (https://www.baeldung.com/spring-controller-vs-restcontroller) to include both the *@ResponseBody* and the *@Controller* annotations in our class.

They also ensure that the resource will be marshalled and unmarshalled using the correct HTTP converter. Content negotiation will take place to choose which one of the active converters will be used, based mostly on the *Accept* header, although other HTTP headers may be used to determine the representation as well.

6. Mapping the HTTP Response Codes

The status codes of the HTTP response are one of the most important parts of the REST service, and the subject can quickly become very complicated. Getting these right can be what makes or breaks the service.

6.1. Unmapped Requests

It's also a good practice to include the *Allow* HTTP header when returning a *405* to the client, to specify which operations are allowed. This is the standard behavior of Spring MVC and doesn't require any additional configuration.

6.2. Valid Mapped Requests

For any request that does have a mapping, Spring MVC considers the request valid and responds with 200 OK if no other status code is specified otherwise.

It's because of this that the controller declares different *@ResponseStatus* for the *create*, *update* and *delete* actions but not for *get*, which should indeed return the default 200 OK.

6.3. Client Error

In the case of a client error, custom exceptions are defined and mapped to the appropriate error codes.

Simply throwing these exceptions from any of the layers of the web tier will ensure Spring maps the corresponding status code on the HTTP response:

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
public class BadRequestException extends RuntimeException {
   //
}
@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
   //
```

corresponding to REST; if for instance, a DAO/DAL layer exists, it should not use the exceptions directly.

Note also that these are not checked exceptions but runtime exceptions – in line with Spring practices and idioms.

6.4. Using *@ExceptionHandler*

Another option to map custom exceptions on specific status codes is to use the *@ExceptionHandler* annotation in the controller. The problem with that approach is that the annotation only applies to the controller in which it's defined. This means that we need to declares in each controller individually.

Of course, there are more ways to handle errors (https://www.baeldung.com/exception-handling-for-rest-with-spring) in both Spring and Spring Boot that offer more flexibility.

7. Additional Maven Dependencies

In addition to the *spring-webmvc* dependency required for the standard web application (/spring-with-maven#mvc), we'll need to set up content marshalling and unmarshalling for the REST API:

```
<dependencies>
   <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.9.8</version>
   </dependency>
   <dependency>
      <groupId>javax.xml.bind</groupId>
      <artifactId>jaxb-api</artifactId>
      <version>2.3.1</version>
      <scope>runtime</scope>
   </dependency>
</dependencies>
```

These are the libraries used to convert the representation of the REST resource to either JSON or XML.

7.1. Using Spring Boot

X

If we want to retrieve JSON-formatted resources, Spring Boot provides support for different libraries, namely Jackson, Gson and JSON-B.

Auto-configuration is carried out by just including any of the mapping libraries in the classpath.

Usually, if we're developing a web application, **we'll just add the *spring-boot-starter-web* dependency and rely on it to include all the necessary artifacts to our project**:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.1.2.RELEASE</version>
</dependency>
```

Spring Boot uses Jackson by default.

If we want to serialize our resources in an XML format, we'll have to add the Jackson XML extension (*jackson-dataformat-xml*) to our dependencies, or fallback to the JAXB implementation (provided by default in the JDK) by using the *@XmlRootElement* annotation on our resource.

X

8. Conclusion

This tutorial illustrated how to implement and configure a REST Service using Spring and Java-based configuration.

In the next articles of the series, I will focus on Discoverability of the API (/restful-web-service-discoverability), advanced content negotiation and working with additional representations of a *Resource.*

All the code of this article is available over on Github (https://github.com/eugenp/tutorials/tree/master/spring-boot-rest). This is a Maven-based project, so it should be easy to import and run as it is.

X

12 COMMENTS Oldest

Alejandro Imass (http://www.yabarana.com) 5 years ago

Hi Eugen, excellent article and one of the few solid references on Spring REST (post 3.x) where you can tell the author knows what he's talking about. Did you ever post the advanced content negotiation ones?

-1

Eugen Paraschiv (https://www.baeldung.com/) 5 years ago

Reply to Alejandro Imass

Hey Alejandro – no, I didn't end up writing on content negotiation – mainly because it's a subject few people are interested in. I usually tackle stuff in the order I see interest from readers (limited time, article TODO list a mile long) – so I can bump this one if you'd like some

more in-depth article on the subject. The plan was to do a custom media type.
And if you'd like to read a bit more on content negotatiation, I touched on the subject here (https://www.baeldung.com/rest-versioning) as well as for testing here (https://www.baeldung.com/2013/01/18/testing-rest-with-multiple-mime-types/). Hope this helps. Cheers,
Eugen.

-1

Alejandro Imass (http://www.yabarana.com) 5 years ago

Reply to Eugen Paraschiv

No, just curious because I stumbled on your article as I was looking for a solution in order to tweak the way that Spring 4 negotiates content with the container. I wound up debugging the actual Spring 4.0.5 to figure out what was going on and Spring uses this order to decided what is the response content type: (1) path extension (.com, .png, etc.) (2) via a query parameter ("format" is default name for this param) (3) from the Accept header. I would have assumed that Accept header should take precedence, after all it's the client's User Agent's requirement, but… Read more »

0

Eugen Paraschiv (https://www.baeldung.com/) 5 years ago

Reply to Alejandro Imass

The difference between Tomcat and Jetty is good to know. I usually stay away from adding media type semantics to the URI of the resource, which is probably why I was lucky enough to not run into this particular bug. However, Jackson issues – especially during content negotiation – are plenty – and debugging Spring itself is really the only valid way of dealing with them. It's also a very good exercise to learn the platform. There are set places where you can easily catch the exception (for example in the DispatcherServlet) – and then you can do a second… Read more »

0

X

Alejandro Imass (http://www.yabarana.com) 5 years ago

Reply to Eugen Paraschiv

"I usually stay away from adding media type semantics to the URI of the resource," I did not design the API (just re-wrote it in Spring) but I agree it's an awful design. It should have been at the very least a query parameter to avoid media type confusion. Many people say they are "RESTful" just because the use minimal HTTP semantics such as GET and POST without even understanding the mere basics of the representational state transfer architectural style. I plan to re-design this API completely in the future and get rid of crap like this but in the… Read more »

+ 0 −

Vidya sagar 4 years ago

Reply to Alejandro Imass

I ran into the same issue,i am not able to solve even after adding above code.surprisingly i am getting "500 Internal Server Error" as my response.

+ 0 −

pranish (http://www.techinfinite.tk) 3 years ago

I have downloaded your code and run as it is on eclipse and tomcat, but it gave me 404 error. I tested it on eclipse browser, postman and chrome but didnt get the required output

+ 0 −

Eugen Paraschiv (https://www.baeldung.com/) 3 years ago

Reply to pranish

Hey Pranish – I just updated the article to point to the more up-to-date version of the code; that should be fully functional.
Let me know if you run into any issues there. Cheers,
Eugen.

+ 0 −

X

karthik 5 years ago

Hi to all , I am new to Spring Boot ,i was face this problem of itteration but it's resolved by using Jackson annotaions of @JsonBackReference and @JsonManagedReference and @JsonIdentityInfo after added these annotaions and bean configuration is everything fine .. but after i could not able to post the data as Json to a controller .Please suggest answer i am facing issue like 1)Requested URL is http://localhost:8080/custMast/state (POST) that time 405 method is coming and JSON is { "custmastCountry" : { "id" : 1, "name" : "INDIA", "customerAddresses" : [ ] }, "name" : "MahaRashtra", "createdOn" : null, "updatedOn"… Read more »

+ 0 −

Eugen Paraschiv (https://www.baeldung.com/) 5 years ago

Reply to karthik

If you have multiple @JacksonBackReference – provide a reference name to each. Hope this helps. Cheers,
Eugen.

+ 0 −

Rai Singh 5 years ago

Hi! Love your articles and your tutorials are top notch.
Question…
From your spring-security-rest-full github project is the following abstract class and interface (https://github.com/eugenp/tutorials/blob/master/spring-rest-full/src/main/java/org/baeldung/persistence/service/common/AbstractService.java) it implements your style of pushing custom persistence-related methods up to the service layer versus creating custom behaviors globally across repositories at the persistence layer as is shown here (https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/#repositories.customize-base-repository)? If so, do you

prefer this method for any particular reason?

Hope my question makes sense.

Thanks,

Rai

+ 0 −

Eugen Paraschiv (https://www.baeldung.com/) 5 years ago

Reply to Rai Singh

Hey Rai – my view on this is that it's pretty much the same thing – since at the end, you have a single definition application to your entire API – so I don't have any strong preference towards one or the other, as long as you only have a single definition and so no duplication. Cheers, Eugen.

+ 0 −

Comments are closed on this article!

X

ezoic (https://www.ezoic.com/what-is-ezoic/)

report this ad

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT-SIDE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

X

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)

JACKSON JSON TUTORIAL (/JACKSON)

HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)

REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)

SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)

SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)

THE COURSES (HTTPS://COURSES.BAELDUNG.COM)

JOBS (/TAG/ACTIVE-JOB/)

THE FULL ARCHIVE (/FULL_ARCHIVE)

WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)

EDITORS (/EDITORS)

OUR PARTNERS (/PARTNERS)

ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)

X