# How to Get Started and Build a Microservice

## Introduction

### Overview

I believe that the best way to learn a new framework or language is to build applications using it.

I also assume that you have some experience with Java, Maven and the IDE of your choice.

For learning we are going to develop a commenting system.

**What will the application do?**

store comments for products or pages

check comments for spam and flag them accordingly.

provide a restful interface to add, delete and retrieve the comments.

**What we will build**

* a persistence tier using Spring Data JPA
* a service Layer using our previous build storage and integrating a legacy module for spam detection
* a restful like API using Spring MVC
* health and metric endpoints for your app
* a command line test client in Java
* Tests :-)

### What You Will Need

* Java 7+
* Maven (3.2+)
* Spring Tools Suite (STS; Optional)

## **What is Spring Boot?**

### Overview

Spring Boot is a new Framework with an opinionated view of building production-ready applications using the standard Spring Framework. Building applications with the Spring Framework used to be a tedious task; especially when starting a new project. Spring consists of multiple modules which you can use individually or integrate into one another. You always had to do either on your own manually. Also, in many cases, modules were not shipped with default configurations of out the box. You had to connect everything by yourself, be it in XML or with Java annotations. That changed with Spring Boot.

Spring Boot is a way to start new applications and use the world of Spring modules with defaults which make sense.

**Its Features:**

* Create stand-alone Spring applications; No need for a Servlet Containers or application server - no WAR files.
* Develop Web apps using embedded Tomcat, Jetty or Undertow
* Provide sense making ‘starter’ poms to simplify your Maven configuration
* Automatically configure Spring as much as possible
* Production-ready addons such as metrics, health checks, and externalized configuration
* No code generation and no requirement for XML configuration - but you can use them, which is important when working with legacy code which itself uses the classic Spring Framework.

### Getting Started

The easiest way to start a new project is by using the [Spring Initializer](https://start.spring.io/)

The second way is to create a Maven pom.xml file. The pom.xml (referenced from now on as pom) is the instruction file that will be used to build your project. Open your text editor and add the following:

<?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>cde.springbootbook</groupId>

<artifactId>comment-store</artifactId>

<version>1.0.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.2.RELEASE</version>

</parent>

</project>

Spring Boot provides various "Starters" for integrating and auto-configuring other modules of the Spring framework, like the embedded Tomcat. We just need to add the starters as a dependency in the pom file and Spring Boot will automatically configure it using default values.

For a web application, we will need the ***spring-boot-starter-web*** which will embed a Tomcat and Spring MVC and configures everything so you can just write your Spring MVC Controllers.

Let's add it to the pom.

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

The next step is to tell Spring Boot how to initialize our application. A Spring Boot application is in its essence a simple Java program with a main method. In it, we need to use the SpringApplication entry point and start it by calling the *run* method. This will bootstrap our application and sprinkle everything with Spring Boot fairy dust aka auto-configure a Tomcat web server and serves our application.

To use the auto-detection and configuration we either need to add the *@EnableAutoConfiguration* annotation to our class or use *@SpringBootApplication* which is a shortcut for multiple annotations which are used commonly.

package de.codeboje.springbootbook.commentstore;

import org.springframework.boot.\*;

import org.springframework.boot.autoconfigure.\*;

import org.springframework.stereotype.\*;

@SpringBootApplication

public class CommentStoreApp {

public static void main(String[] args)

throws Exception {

SpringApplication.run(CommentStoreApp.class, args);

}

}

Now you can run it either as a java application (STS: Run as Java Application) or by using maven with mvn spring-boot:run. Of course, it doesn't do anything right now; so let's add a simple web endpoint.

Set the *@RestController* on the *CommentStoreApp* class from above and add the method

@RequestMapping("/")

String home() {

return "Hello World!";

}

The Annotation will mark our class as a Spring MVC Controller providing a restful endpoint. Spring Boot recognizes it and will setup Spring MVC and a default Json Transformation using the Jackson library.

The *RequestMapping* Annotation on the new method provides routing information for Spring. The root path of our application is tied to this method, i.e. when you later access the application in the browser, Spring will know it must call our *home* method for fulfilling the request. We will cover this more in another chapter.

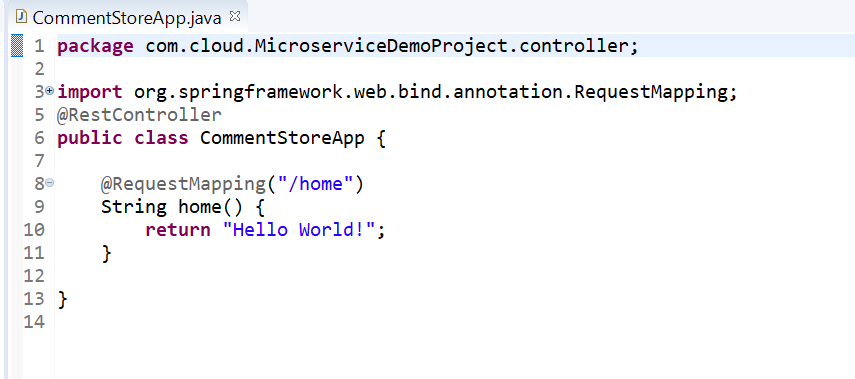
Run the example again and open your web browser to localhost:8080 and you should see a Hello World!.

To exit the application hit ctrl-c.

Voila, your first Spring Boot app.

But that's only a toy application, so with the next chapter, we will start our sample application.

Create a maven project with maven and add the dependency -web.



Run the example again and open your web browser to localhost:8080 and you should see a Hello World!.

## **Data Access Layer with Spring Data JPA**

**we are going to implement our persistence layer**.

For that, we are going to use the **Spring Data JPA Framework.**

Spring Data is an umbrella project to make working with **data stores easier** and **encapsulating the actual data storage access**; meaning you could switch the database backend without changing a single line of your data access code.

To use it in our application we need to add the dependency to the spring boot starter.

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-data-jpa</artifactId>**

**</dependency>**

When you use **H2, HSQL or Derby** as an embedded database

Spring Boot takes care of setting up the whole access layer, whenever we use a different database we must define access to the database.

We set those configurations in a properties file named ***application.properties*.**

Create the file under src/main/resources and add the following section.

spring.datasource.url=jdbc:h2:mem:mydb

spring.datasource.username=sa

*spring.datasource.url*

* tells Spring Boot where the database is located and which driver to use. It follows standard JDBC URL naming scheme.

*spring.datasource.username*

* is the username to access the database.

In the case of an in-memory DB, we do not need a password, but it would be defined with the property *spring.datasource.password*.

In case we would use Postgres as the backend, the previous section could look like

spring.datasource.url=jdbc:postgresql://localhost:5432/commentstore

spring.datasource.username=postgres

spring.datasource.password=password

When we start our app now, database access is automatically configured and ready to go. The only thing missing is the database itself, but we can let hibernate create it by adding the following to our *application.properties*.

**spring.jpa.hibernate.ddl-auto=update**

The property *spring.jpa.hibernate.ddl-auto* defines if and when hibernate will create our database with all tables; in our case we let it update the schema every time we start the application.

### **Creating Our Entity**

 in our case, ***CommentStoreApp*,** will be scanned for classes annotated with *@Entity, @Embeddable* or *@MappedSuperclass*.

If they reside in a different package, you must add ***@EntityScan*** to your configuration class.

As we have placed the model class in a different package (on purpose), we need to insert the following line to our *CommentStoreApp* configuration.

@EntityScan(basePackages= {

"de.codeboje.springbootbook"

})

Spring Boot now scans all packages starting from *de.codeboje.springbootbook* for our entity classes. So create the class *CommentModel* in the package *de.codeboje.springbootbook.model*. Imports and get/setter are omitted for readability.

@Entity

@Table(

name = "comments\_model",

indexes = {

@Index(name = "idx\_pageId",

columnList = "pageId"

)

}

)

@TypeDefs({

@TypeDef(name = "calendarUTC",

typeClass = UtcCalendarType.class,

defaultForType = Calendar.class)

})

public class CommentModel implements Serializable {

@Id

@Column(length = 36)

private String id;

@Version

private Integer version;

@Temporal(TemporalType.TIMESTAMP)

@Type(type = "calendarUTC")

private Calendar lastModificationDate;

@Temporal(TemporalType.TIMESTAMP)

@Type(type = "calendarUTC")

private Calendar creationDate;

@Column(length = 32)

private String pageId;

@Column(length = 32)

private String username;

@Column(length = 32)

private String emailAddress;

@Column

private boolean spam;

*@Entity*: Tells Hibernate and Spring that this is our Entity class.

*@Table*: defines a table name and sets an index on the pageId;

we will query later on it *@TypeDefs*: Defines a Hibernate type transformation, so we store and handle our dates in UTC.

If we do not set it as fixed here, it will depend on the locales used through your systems and can end in a mess. I omit the code here as it is not essential for building the microservice.

*@Id*: Unique ID of our *CommentModel* record

*@Version*: A version ID used by Hibernate for optimistic locking.

I think the others fields are pretty self-explanatory so we leave them as is.

When we would start now our application, Spring Boot will set up our Database access, detecting our Entity class, creating our database and table. We are ready to use our model

### **Storing and Retrieving Data Using the Spring Data JPA Repositories**

As I mentioned earlier, Spring Data offers a common layer to work with our data storage. It hides away the specific implementations of JPA or Hibernate and provides access through *Repository* interfaces.

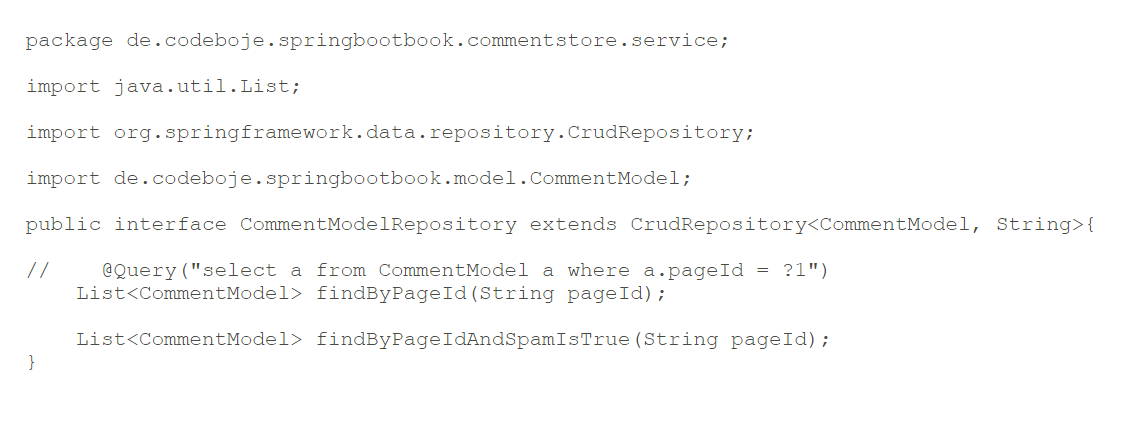
During startup Spring will create classes implementing our *Repository* interface, providing all the logic to work with our data and making those available in the Spring context. We can use them like any other regular Spring Bean - an object managed by the Spring Framework.

In your repository interface, you can define your queries either by a naming convention for the method name or explicit with the Spring Data Query annotation *@Query*.

Additionally, there are some sub-classes with more predefined methods, i.e. the CrudRepository offering support for basic Create-Retrieve-Update-Delete Operations.

By default all packages below the configuration class, in our case *CommentStoreApp*, are scanned for *Repository* interfaces.

Create our *CommentModelRepository* in [*de.codeboje.springbootbook.commentstore.service*](http://de.codeboje.springbootbook.commentstore.service/) and extend the *CrudRepository* interface.:



package de.codeboje.springbootbook.commentstore.service;

import java.util.List;

import org.springframework.data.repository.CrudRepository;

import de.codeboje.springbootbook.model.CommentModel;

public interface CommentModelRepository extends CrudRepository<CommentModel, String>{

// @Query("select a from CommentModel a where a.pageId = ?1")

List<CommentModel> findByPageId(String pageId);

List<CommentModel> findByPageIdAndSpamIsTrue(String pageId);

}

We use the naming scheme approach and tell Spring Boot we search for all records with a specific pageId. As mentioned before the same result could be achieved with an @Query annotation like:

@Query("select a from CommentModel a where a.pageId = ?1")

You can even define more complex queries by naming your method. Take a look at the second query type:

findByPageIdAndSpamIsTrue(String pageId)

It will retrieve all comments filtered by pageId, and the spam attribute must be true. You will find a full set of supported keywords in the [query creation section](http://docs.spring.io/spring-data/jpa/docs/1.4.1.RELEASE/reference/html/jpa.repositories.html#jpa.query-methods.query-creation) of the Spring Data JPA documentation. The downside is it creates pretty ugly method names and when you change your query you also need to replace the method name in all classes using this method. With a good IDE, refactoring is fast, but you still might deploy more changes than were actually needed. Also, it can be hard to read sometimes.

To access our data, we can obtain our Repository via Springs dependency injection and can use it without worrying about the whole database backend. We will cover that further in the next session

### **Tips**

When developing on a larger scale, some settings are pretty helpful.

#### See the SQL Sent to Your DB

With more complex models, I often had the need to see the SQL query Spring actually builds and sends to the database. You can see it by adding

spring.jpa.show-sql=true

to your *application.properties*. The statements are logged out on stdout and/or your configured logging.

#### Preventing Errors on Dead DB Connections

The underlying logic usually accesses your database with some kind of connection pool. If your code needs to send a query, it gets a connection from the pool and when it is done it gives it back. However, what can happen due to network issues, DB timeouts, etc., is that the connection got stale. If the connection stale it is essentially dead and can not be used. Typically, the pool does not clean them up, and the next time you use a connection, it might hand you the broken one. However, you can tell the pool to check each connection before using it.

To enable that add the following section to your *application.properties*

spring.datasource.test-on-borrow=true

spring.datasource.validation-query=SELECT 1

*spring.datasource.test-on-borrow* enables the check for valid connections *spring.datasource.validation-query* defines a query for the check.

The check does come with a drawback, though. A minor loss in performance. Depending on your particular scenario it might be noticeable.

### Recap

Before we continue, let's review what we have covered and check your understanding with a short quiz.

* Which Start do we use for Spring Data JPA?
* What steps are needed for using an H2 DB?
* Where do you configure the database URL and credentials?
* Where does Spring Boot look by default for JPA Entity classes?
* How can you specify a model in a different package?
* How can you see the SQL going to your database?

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

**Implementing the Service Layer**

we are going to implement our service layer. It will store and retrieve our data using the repository we create and it integrates a legacy module using Springs XML configuration

### Our Service Interface

Create the *CommentService* interface in the [*de.codeboje.springbootbook.commentstore.service*](http://de.codeboje.springbootbook.commentstore.service/) package and paste the following code:

public interface CommentService {

String put(CommentModel model) throws IOException;

List<CommentModel> list(String pageId) throws IOException;

CommentModel get(String id);

List<CommentModel> listSpamComments(String pageId) throws IOException;

void delete(String id);

}

It defines a method for storing a comment *put*, to delete a comment *delete*, retrieve a single comment *get*, and getting a list of comments *list* a specific page.

#### **Our Service Class**

Our Service will use the legacy module, which implements a simple span detection, so we add it first as a dependency to the pom.

<dependency>

<groupId>de.codeboje.springbootbook</groupId>

<artifactId>spam-detection</artifactId>

<version>1.0.0-SNAPSHOT</version>

</dependency>

The legacy module is not in a public Maven repository, so you need to check out the code and install it first by executing mvn clean install

The next step is to create our service class implementing our previously defined interface.

@Service

public class CommentServiceImpl implements CommentService {

@Autowired

private SpamDetector spamDetector;

@Autowired

private CommentModelRepository repository;

}

First, we tell Spring with the *@Service* annotation that this is a service class and it should be initialized and made available for the others components.

Next, we create a variable for our *CommentModelRepository* as we are going to access data.

The *@Autowired* annotation tells Spring to auto wire this field, so it will look for ONE instance in its context which implements the *CommentModelRepository*.

The actual implementation of the interface is created during startup by Spring Data.

We do the same with our legacy module. It provides exactly one interface, *SpamDetector*. If we started the application now, Spring would complain that it does not find a class implementing the *SpamDetector* interface.

#### **Adding a Legacy Module with its Own Spring XML Configuration**

Our legacy module will check for spam words in a text. The words are defined in a plain text file with each unwanted word on its own line.

The legacy module uses its own Spring XML configuration where it is setup and ready to use as a black box.

It includes a bean definition for our *SpamDetector* and also makes use of a spring property, so we need to set this one up too. Remember this is a stripped-down version for demonstration purposes, in real world cases, they are more complex.

[Larger View](https://viewer.books24x7.com/assetviewer.aspx?bookid=128249&chunkid=385161748&resumebookmarkid=121e57fb-5cb6-e811-aa3a-005056957207)

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns= "http://www.springframework.org/schema/beans"

xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation= "http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="legacySpamDetector" class=

"de.codeboje.springbootbook.spamdetection.impl.SimpleSpamDetector">

<constructor-arg>

<value>${sbb.spamwords.filename}</value>

</constructor-arg>

</bean>

</beans>

The configuration is packed in the jar file of the legacy module so that we can use it directly.

Open our *CommentStoreApp* and add:

@ImportResource(value={"classpath\*:legacy-context.xml"})

to the class. The *@ImportResource* will validate the mentioned context and include it in the current one thus making all the beans in it available.

By default, it will look in the current project, but with prepending *classpath\** Spring searches the whole classpath for our file at the root position

If you started the application now, Spring would find the context and try to initialize the bean but will ultimately fail because we have not provided a value for the Spring

Property *sbb.spamwords.filename* yet.

To do so, add the following to our *application.properties*

sbb.spamwords.filename=src/main/resources/words.spam

and our application should start now without problems.

#### **Adding a Comment**

Our *CommentService* interface provides one method for adding or updating a new entry. The caller will send a full *CommentModel*, and we need to decide if we create a new entry or update an existing one. The entry is identified by the ID property.

First, we need to check and retrieve an existing entry. As we will work with the retrieved entry anyways and it is pretty small, we directly load it.

To load an entry by Id, we use the *findOne* method provided by our repository; it is one of the methods provided by the *CrudRepositry* and takes exactly one argument - the ID of the record.

repository.findOne(id)

Implement it as the get method and add

@Override

public CommentModel get(String id) {

return repository.findOne(id);

}

to our service implementation.

If Spring finds an existing record it will return the instance; otherwise, it will return null. Based on this we can either merge the new and existing record or create a new one.

Creating a new one is as simple as:

repository.save(model);

Spring takes the model class and serializes it to the database; overriding any existing record with the new data. Except when the version property differs. Then it will throw an exception that the DB record was modified - optimistic locking.

Our method for adding or updating looks like

@Override

@Transactional

public String put(CommentModel model)

throws IOException {

if (StringUtils.isEmpty(model.getId())) {

model.setId(UUID.randomUUID().toString());

}

if(spamDetector.containsSpam(model.getUsername())

||

spamDetector.containsSpam(model.getEmailAddress())

||

spamDetector.containsSpam(model.getComment())) {

model.setSpam(true);

}

final CommentModel dbModel = get(model.getId());

if (dbModel != null) {

dbModel.setUsername(model.getUsername());

dbModel.setComment(model.getComment());

dbModel.setLastModificationDate(

Calendar.getInstance());

repository.save(dbModel);

}

else {

model.setCreationDate(Calendar.getInstance());

model.setLastModificationDate(

Calendar.getInstance());

repository.save(model);

}

return model.getId();

}

If the comment already has an ID, our service will update it. If it does not have one, we will create the ID by ourselves. Next, we check the *username, emailAddress* and *comment* fields for spam and if the service detects any, the comment is marked as spam.

In either case, for storing our DB model, we use the repository.save method.

#### Transaction Management

When working with complex models or different backends you usually need some transaction handling. So if anything goes wrong in one backend, you can roll back your data in all.

The Spring Family also provides a module for transaction management *Spring Transaction Management.*.

It is used in Spring Data, and we can enable the basic version in two simple steps.

First, annotate your method with *@Transactional* like in our version above and second, enable it in the Spring Boot Configuration by adding the *@EnableTransactionManagement*annotation to *CommentStoreApp*.

Spring will wrap your class in a proxy and add the transaction handling. If your method fails with an exception, Spring will initiate a rollback on the transaction manager. By default, Spring Data is setup automatically in Spring Boot to support this when we enable transaction handling with the *@EnableTransactionManagement* annotation.

#### Writing Tests for the Service

Spring Boot provides an easy starter dependency which includes Spring Boot Test modules and a handful of useful libraries like JUnit, Hamcrest, and co.

Add the following section to your pom, and we can start to write a test.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

The Spring Test modules use JUnit at its core but enhance it to make use of the dependency injections of Spring.

Create *CommentstoreServiceImplTest* in src/test/java and let's start with implementing a test.

First, we need to tell JUnit that we use the Spring Feature

@RunWith(SpringRunner.class)

Second, we must set up our Spring Boot application for testing. The easiest way is to annotate the class with

@SpringBootTest()

The Spring Boot test feature now auto configures our application completely as it would run later but also induces some testing functions. So, any Spring Bean, either set up by your configuration or found with a *ComponentScan*, will be available in the Spring Context. If your application uses a lot, it might slow down your test runs.

But, if your DAO or service does not have any dependencies except JPA, you can use the *@DataJpaTest* annotation instead. Spring Boot will now only scan for *@Entity* classes and configure Spring Data JPA repositories. All other *@Component* beans are ignored for the *ApplicationContext* of this test.

Be aware though, that certain features like the *@Value* injection are only set up by using the *@SpringBootTest*, or you have to do it manually.

Now that our test is initialized through Spring, and we can use dependency injection in our test, i.e. we can auto wire our *CommentModelRepository*.

For our test, we will set up an in-memory DB and use this for storing the comments. As tests should run independently, we clear our table before each test run. However, we are lazy and use the same model values for the whole test.

@Before

public void setup() {

model = new CommentModel();

model.setUsername("testuser");

model.setId("dqe345e456rf34rw");

model.setPageId("product0815");

model.setEmailAddress("example@example.com");

model.setComment("I am the comment");

repository.deleteAll();

}

Now we can create a JUnit test by annotating the method with *@Test* and implement it using our injected service.

@Test

public void testListNotFound() throws IOException {

service.put(model);

List<CommentModel> r = service.list("sdfgsdwerwert");

assertTrue(r.isEmpty());

}

The rest of the process is basic JUnit testing, and you can use all features that the framework has to offer.

### Recap

Before we continue, let's review what we have covered and check your understanding with a short quiz.

* How can you include an existing Spring Configuration XML File?
* How do you make your service class available to Spring?
* Where are Spring Properties configured?
* What steps must you take to include Transaction management?
* Can you use Spring dependency injection in JUnit tests? How do you set it up?

In the next chapter, we are going to expose our service in a Rest API.

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

## Implementing the Rest API

### Overview

we are going to implement a Rest API for our service using Spring MVC.

Spring MVC is Spring Framework' answer for developing web applications. It is based around a central *DispatcherServlet*, forwarding each request to the correct handler to handle that request.

We will separate our API in two use cases and classes.

* Retrieving comments
* Writing and deleting comments

A typical REST API makes an instance of your model under its own URL. Interaction, like retrieving or deleting, is all done using the same endpoint, but with different HTTP request methods.

* GET for retrieving
* POST for creating
* PUT for changing
* DELETE for deleting

In our sample application, we will use the single endpoint approach for creating comments and deleting them under their own endpoint. Retrieving a single comment does not make sense here.

Thus we are providing an endpoint for retrieving all comments of a particular page.

To use Spring MVC in our Spring Boot application, we must add the Starter to our pom file.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

### Rest API Implementation

In Spring MVC the main component we need to implement is a controller class. The dispatching mechanism in Spring MVC will eventually call a method on your controller class to fulfill the request.

Controller classes are plain Java Beans which are annotated with the *@Controller* Annotation.

Spring MVC scans during startup for classes with the *@Controller* annotation and will register them.

Next, we need to inform Spring MVC which method in our controller maps to which endpoint in our web application. We can do this with *@RequestMapping* annotation on a method.

That is the basic theory, let's dive into the actual implementation.

#### Write Operations

Create the new class *WriteController* in the [*de.codeboje.springbootbook.commentstore.restapi*](http://de.codeboje.springbootbook.commentstore.restapi/) package and then add the *@Controller* annotation on the class level. Moreover, while we are at it, we directly add our *CommentService* service and let Spring auto wire it.

@Controller

public class WriteController {

@Autowired

private CommentService service;

}

In the next step, we declare our first endpoint. We post a new comment in our sample application with an HTTP Form Post.

With the *@RequestMapping* Spring knows that this is an endpoint. The *value* is the path under which this method is made available, and the *method* defines that we only accept HTTP Post request.

MVC uses both values for dispatching an incoming request. If for example, someone sends a GET request to our endpoint, MVC will not route it to our method and return a 400 Bad Request error.

MVC provides a way to let it map the incoming form fields to method parameters. To enabled it, we annotate each of our parameters with *@RequestParam*. The parameter of the annotation defines the name of the field in the submitted form. Regular values are submitted as strings.

Hint: The MVC mapping mechanism can convert more as we use here in our sample application

@RequestMapping(value = "/create",

method = RequestMethod.POST)

@ResponseStatus(value = HttpStatus.OK)

public @ResponseBody String create(

@RequestParam("comment") final String comment,

@RequestParam("pageId") final String pageId,

@RequestParam("emailAddress") final String email,

@RequestParam("username") final String username)

throws IOException {

CommentModel model = new CommentModel();

model.setPageId(pageId);

model.setEmailAddress(email);

model.setComment(comment);

model.setUsername(username);

String id = service.put(model);

return id;

}

Normally, when we return a String value in our method, Spring MVC will look up for a view aka template to forward the request to. Depending on your configuration you can use plain old JSPs or any of the newer template engines for that. You can find more about it [here](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/view.html).

However, in our case we just want to return the ID of the created comment, so we will use the *@ResponseBody* annotation on the return value. MVC will now just return our object here as a plain string. In which format the object is returned is determined by a *ContentNegotationStrategy* and the actual conversion is done by an HTTP message converter, i.e. JSON.

The *@RequestMapping* has a *produces* attribute that defines which format the endpoint will return. Using the values of the *@RequestMapping*annotation and by analyzing the request Spring MVC decides which converter to use. To keep it simple here, in our case Spring will fallback to plain String conversion.

Our method now creates an instance of our *CommentModel* and fills it with the values from the form.

Afterward, it calls our service to add it and return the ID of the comment.

##### Integration Testing

How can we confirm that it works? By testing it and we will start with a Unit test.

Create the *WriteControllerTest* under *src/test/java* in the package [*de.codeboje.springbootbook.commentstore.restapi*](http://de.codeboje.springbootbook.commentstore.restapi/). Now we configure it with *@RunWith* and *@SpringBootTest* like so Spring Test has the right context.

The test could run now but would start an embedded servlet container, just as in a regular application start. However, depending on your application and your test runtime environment it might be better to test only in a mock web environment. To enable it, we only need to add *webEnvironment = WebEnvironment.MOCK* to the *@SpringBootTest* annotation. Now Spring Boot Test runs our test with a mock web environment, and we can test using Spring MVC Test.

The main helper of Spring MVC Test is *MockMvc*, and it provides methods to interact with *Controller* via the mock web environment. In earlier versions of Spring Boot, you had to set this one up manually, but now we just need to add the *@AutoConfigureMockMvc* annotation to the test, and voila, *MockMvc* can be injected like any other regular Spring Bean and is ready to use.

When you are going to test only the *Controller*, you could also use *@WebMvcTest* instead of *@SpringBootTest* as an alternative. This will only set up your web configuration for an MVC mock environment and ignores all other *@Component* beans. To provide mocks for depending service classes, you can use the *@MockBean* annotation feature in the test class. Simply set it on a field in the test like

@MockBean

private CommentService service;

and Spring Boot Test provides you a Mockito mock for it. Besides, it also resets the mock after each test method.

We will write an integration [test for the *WriteController*](https://viewer.books24x7.com/assetviewer.aspx?bookid=128249&chunkid=507270991&resumebookmarkid=2220df36-4db6-e811-aa3a-005056957207) and use the *@WebMvcTest* feature later for testing the *ReadController*.

Let's start.

@SpringBootTest(webEnvironment = WebEnvironment.MOCK)

@AutoConfigureMockMvc

public class WriteControllerTest {

@Autowired

private WebApplicationContext context;

@Autowired

private CommentService service;

@Autowired

private MockMvc mvc;

}

Now it is ready to use, and we can create our first test. Create the test method and add the *@Test* annotation from JUnit.

To call a web method on the mock we use the perform method. It needs a *RequestBuilder* which knows how to build a specific request, in our case one for form posts. *MockMvcRequestBuilders.post* builds it and will point it at the */create* endpoint.

Now on the *RequestBuilder* we can set our form fields with the *param* method.

When called, *perform* will now execute our request in the Spring Test black box.

The *perform* method also returns a *ResultActions* where we can chain certain test and validate if the request was successful.

andExpect(status().is(200))

Read it as "We expect that the result has a status code of 200".

As we want to check that our comment is really created, we will load it directly via the *CommentModelRepository*. But for that, we require the ID returned by the REST API. To access it Spring Test provides the *andReturn()* *ResultActions*, which will return a class for working with the result.

To retrieve the ID as a plain String, we get the actual response from the result and use the *getContentAsString* on this.

result.getResponse().getContentAsString()

As we only return one single string, we are ready to use it as it is.

@Test

public void testPost() throws Exception {

CommentModel model = setupDummyModel();

MvcResult result = this.mvc.perform(

MockMvcRequestBuilders.post("/create")

.param("comment", model.getComment())

.param("pageId", model.getPageId())

.param("emailAddress", model.getEmailAddress())

.param("username", model.getUsername())

)

.andExpect(status().is(200))

.andReturn();

String id = result.getResponse()

.getContentAsString();

CommentModel dbModel = service.get(id);

assertNotNull(dbModel);

assertEquals(model.getComment(),

dbModel.getComment());

assertEquals(model.getPageId(),

dbModel.getPageId());

assertEquals(model.getUsername(),

dbModel.getUsername());

assertEquals(model.getEmailAddress(),

dbModel.getEmailAddress());

assertNotNull(dbModel.getLastModificationDate());

assertNotNull(dbModel.getCreationDate());

assertFalse(model.isSpam());

}

To verify that our Rest API and service worked, we retrieve the newly created model and compare them to our input. There should be no difference.

In this section, we used an integration test which made more sense than just testing the Rest API methods. Of course, you can do pure Unit testing and provide a mock object of our service class using Mockito or similar. That is up to you and your product.

#### Read Operations

Create the new class *ReadController* in the [*de.codeboje.springbootbook.commentstore.restapi*](http://de.codeboje.springbootbook.commentstore.restapi/) package and then add the *@RestController* annotation on the class level. Moreover, while we are at it, we can directly add our *CommentService* service and let Spring auto wire it.

The *@RestController* extends the regular *@Controller* annotation and triggers some additional configurations in Spring MVC. Our endpoint will be set up as a Rest endpoint and by default accepting JSON requests and returning JSON as results.

*@RestController* is a shortcut and you can archive the same result by using the regular *@Controller* annotation and adding @ResponseBody to your service method and setting the *produces* to *application/json”* on the *@RequestMapping* annotation of your method.

@RestController

public class ReadController {

@Autowired

private CommentService service;

}

By default Spring MVC uses the Jackson library for mapping from JSON to Java object and vice versa.

##### Retrieving Comments

@RequestMapping(value = "/list/{id}")

public List<CommentDTO> getComments(@PathVariable(value = "id") String pageId)

throws IOException {

List<CommentModel> r = service.list(pageId);

if (r.isEmpty()) {

throw new FileNotFoundException("/list/" + pageId);

}

return transformToDTO(r);

}

Our endpoint accepts a value in its path, and we will map this one to our method parameter by using the *@PathVariable* annotation. It is similar to the *@RequestParam* annotation we used before but looks up the value in the path of our endpoint and not in the request's parameters. Note: a *@PathVariable* is mandatory. If you send a request to that endpoint without the PathVariable, it is a different endpoint for Spring.

The method itself simply looks up the comment and transforms it to the data transfer object (DTO) we use on the interface.

HINT: We could use your business model *CommentModel* here directly as it is a regular Java Bean BUT it does have a drawback. Whatever component is using our class will also require all its dependencies; otherwise, it can be loaded by the Java Virtual Machine. So each application using our model class will also need all the JPA and Spring annotations we used and due to using our own *UtcCalendarType* we also have a dependency on Hibernate now. This dependency clutter can cause trouble and strange side effects. Be aware and choose wisely.

##### Testing Using @WebMvcTest

As mentioned before in the testing the *WriteController* section, we are going to implement the tests for the *ReadController* using the *@WebMvcTest*annotation and pure mocking.

Let's start.

Create the *ReadControllerTest* under *src/test/java* in the package [*de.codeboje.springbootbook.commentstore.restapi*](http://de.codeboje.springbootbook.commentstore.restapi/) and add the following annotations to the class:

@RunWith(SpringRunner.class)

@WebMvcTest(ReadController.class)

@AutoConfigureMockMvc

@MockBean({SpamDetector.class, CounterService.class})

The first tells JUnit to use the SpringRunner, so we have Spring support in the test. It is the same as before. The annotations *@WebMvcTest* and *@AutoConfigureMockMvc* provide us with the mock environment and *@MockBean* will mock for us the SpamDetector, so it will not instantiate the class; it's included via an XML import and not by *ComponentScan*. It also mocks the *CounterService* of Spring Boot Actuator which is covered in the [HealthCheck and Metrics](http://viewer.books24x7.com/assetviewer.aspx?bkid=128249&destid=50" \l "50" \t "_parent) in next session.

Next, we will mock the *CommentService* service as a field (we are going to use it) and inject the *MockMvc* too.

public class ReadControllerTest {

@MockBean

private CommentService commentService;

@Autowired

private MockMvc mvc;

}

Now we can write the test. First, we create a dummy Comment and tell Mockito to return it when someone calls *list* on our *CommentService*.

when(this.commentService.list( anyString())).thenReturn(mockReturn);

When the list method is called on the mock with any given pageId (*anyString()*), it will return a list with our comment.

@Test

public void testGetComments() throws Exception {

CommentModel model = setupDummyModel();

List<CommentModel> mockReturn = new ArrayList<CommentModel>();

mockReturn.add(model);

when(

this.commentService.list(anyString())

).thenReturn(mockReturn);

this.mvc.perform(

get("/list/" + model.getPageId())

)

.andExpect(status().is(200))

.andExpect(

jsonPath("$[0].id",

is(model.getId()))

)

.andExpect(

jsonPath(

"$[0].created",

is(SDF.format(

model.getCreationDate()

.getTime()))

)

)

.andExpect(

jsonPath(

"$[0].username",

is(model.getUsername())

)

)

.andExpect(

jsonPath(

"$[0].comment",

is(model.getComment())

)

)

.andExpect(

jsonPath(

"$[0].email\_address",

is(model.getEmailAddress())

)

);

verify(this.commentService, times(1)

).list( anyString());

}

When we call the *perform* method on the mock, it will run the given request in the mock environment. The result is then checked against our input model. Also, at the end we check that the *list* method was called once on the mock; *verify* and *times* are part of Mockito.

##### Object Mapper

For changing the *ObjectMapper* we must declare a bean in the *CommentStoreApp* class of type *ObjectMapper* and use the *@Primary* annotation to mark it as the primary bean for this type.

@Bean

@Primary

public ObjectMapper initObjectMapper() {

return new CommentstoreObjectMapper();

}

Hint: If you also develop clients for your Rest APIs you should share a common Object mapper for the server and client side. Here in the example application, it is the *CommentstoreObjectMapper*.

##### Exceptions

When a controller method throws an exception, MVC will return a 500 Internal server error with a Stacktrace. We do not want that for our *ReadController*. When no comment is found we want to return a proper 404 - Resource not found.

So let's override the default behavior and define an exception handler. In the *ReadController* class add a new method and set the *@ExceptionHandler*on it. Its parameter must be of type *Exception*, here *FileNotFoundException*.

When our service or controller would now throw the *FileNotFoundException* it will be caught by Spring MVC and forwarded to our *ExceptionHandler*. The *ExceptionHandler* has now the full responsibility to handle the error.

In the sample application, we will only log the Exception for debugging and return to the browser with a 404 using the *@ResponseStatus* annotation. However, you are not limited to a simple return value; you could even return a rendered error page or a JSON describing the error.

@ExceptionHandler(FileNotFoundException.class)

@ResponseStatus(value = HttpStatus.NOT\_FOUND)

public void handle404(Exception ex, Locale locale) {

LOGGER.debug("Resource not found {}", ex.getMessage());

}

### Securing Our App

In the current state, our application is open to everyone. If you know the endpoint you can read, update or delete files. Even in an intranet, you do not want to do that.

Luckily, Spring Boot also provides an out of the box solution for securing your application. Two simple steps will give us basic authentication on all HTTP endpoints which is enough for service like this one.

First, add the Spring Security Starter to the pom.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

Second, we want to set our own username and password. Open the *application.properties* and add:

security.user.name=admin

security.user.password=mypassword

This will configure Spring Security to use these as the default credentials.

Start the application and every time you access it now your browser asks for the credentials.

You are not limited to basic authentication and can use the full power of Spring Security including OAuth2, which Spring Boot now even supports better with specific annotations. For further reading, I recommend [this](http://docs.spring.io/spring-boot/docs/1.4.2.RELEASE/reference/htmlsingle/#boot-features-security-oauth2).

When security is enabled, it will also be used in the tests by default. In our simple case, we can turn HTTP Basic Auth off by adding

security.basic.enabled=false

to the application.properties in the *src/test/resources* folder.

If you use any of the Spring Security features for authorization, you can use their test support by adding

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-test</artifactId>

<scope>test</scope>

</dependency>

to your pom. Now you can use all Spring Security Test annotations as you would regularly do.

#### Manual Testing with Postman

[Postman](https://www.getpostman.com/) is a tool for testing APIs available via the HTTP protocol. It is available as a Chrome extension and a Mac and PC application. It is free and a helpful tool. Saved me a few times already.

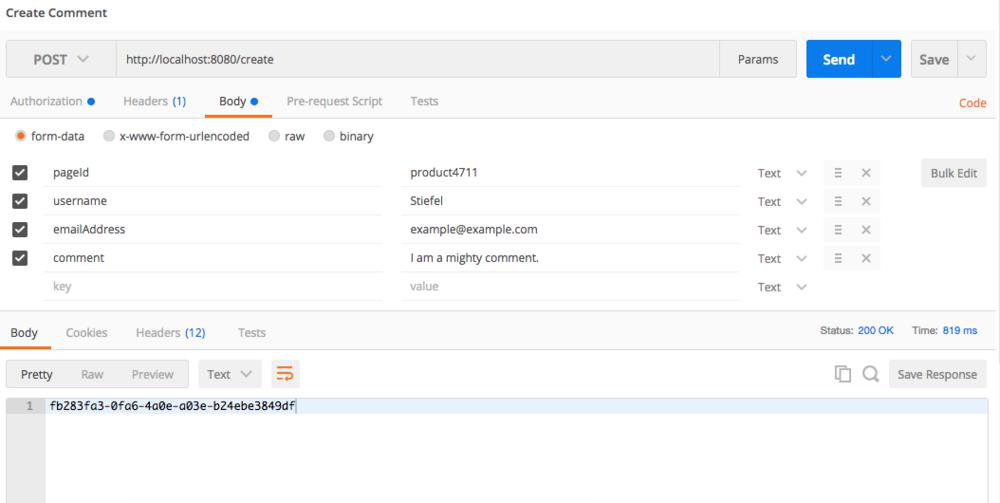
After installing it, import a new collection and point to the File *SpringBootBook.postman\_collection* in the source code of our application. I pre-made the essential requests.

##### Creating a Comment

By default, the spring boot application listens on port 8080.

The endpoint for creating comments is accessible at */create* and accepts a post request. We also need to enter all our required fields, and as we secured our application before, we also need to enter the credentials.

After clicking the send button it will look like this:



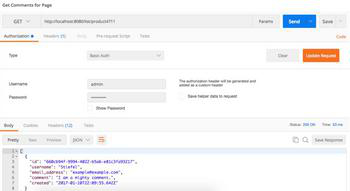
As a result, we receive the ID of our newly created comment.

##### Receiving Comments

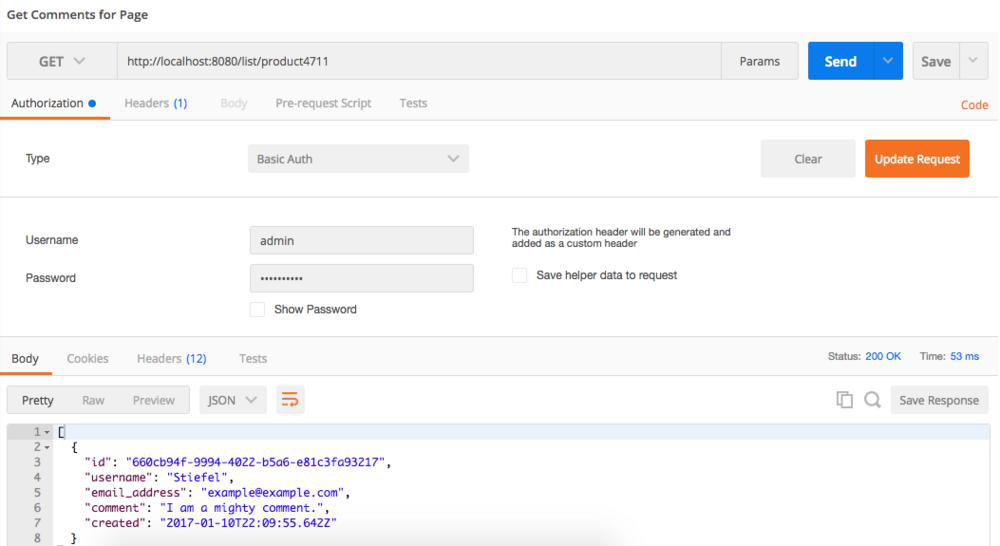
The endpoint for receiving all comments is accessible at */list/* and requires the ID of a page as a request path variable. We will use the pageId of our previously created comments.

Click the send button and you will see something like:

[Larger View](https://viewer.books24x7.com/assetviewer.aspx?bookid=128249&chunkid=217439997&resumebookmarkid=2220df36-4db6-e811-aa3a-005056957207)

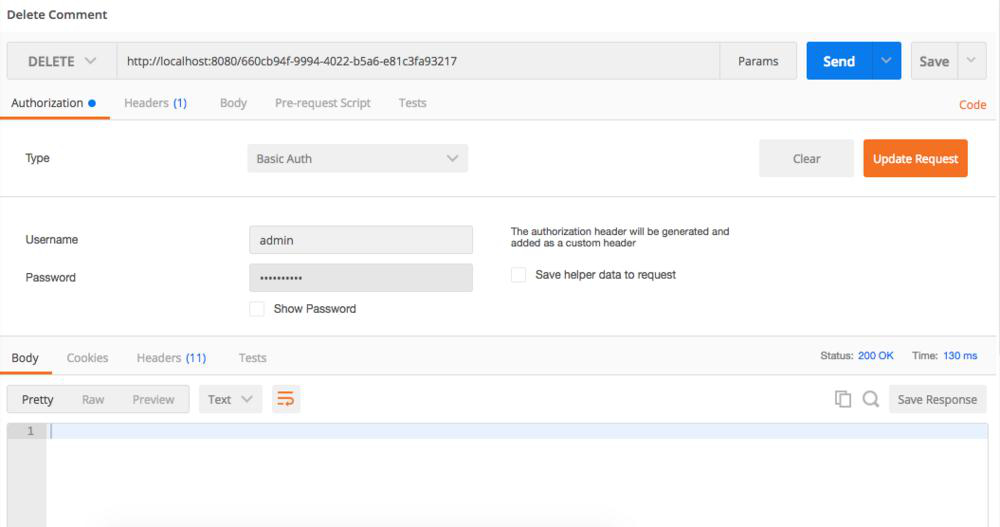


Now create a comment with one of our spam words in any field. Then we will retrieve the spam comments by using our endpoint *listspam* which also expects a page ID in the request path.



##### Deleting a Comment

Now we are going to use the ID of the spam comment for our delete request. The endpoint is available under the root path and requires a comment ID in the path. Remember, each comment is accessible under its own URL and our API allows interaction. To delete the comment, we must change to an HTTP DELETE and just send the request.



You will get a status code of 200, and the comment is deleted. Verify it by resending one of the list requests from before.

### Recap

Before we continue, let's review what we have covered and check your understanding with a short quiz.

* How do you configure a Spring MVC Controller for handling form posts?
* What is the difference between the *Controller* and *RestController* annotation?
* How can you secure your application with Spring Security?
* How can you declare your own ObjectMapper for JSON transformation?
* How do you define a custom handler for Exceptions?
* What does the *ResponseBody* annotation do?
* How can you test Spring MVC controller?

In the next topic, we are going to take a look at logging.

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

## Logging

### Overview

By default, Spring Boot uses Logback for logging. It is also set up to enable routing from common Java logging libraries like Java Util Logging, Commons Logging, Log4J or SLF4J.

In the sample application, we use SLF4J.

To obtain a logger instance we call *getLogger* of the *org.slf4j.LoggerFactory*. The Method is static, and we can create a constant in our class, i.e. *ReadController*

private static final Logger LOGGER =

LoggerFactory.getLogger(ReadController.class);

Now we can use *info, debug* or any of the other logging methods.

By default, our log message is printed on the console. This is not suitable for running the microservices later, so we are going to change it to file based logging.

### Logging to File

Spring Boot offers two properties to enable file logging. They can work together too.

The properties are set in your *application.properties*.

* *logging.file*: Specifies a file to write in; can be a relative or absolute path.
* *logging.path*: Specifies a directory where to store the log file. If *logging.file* contains a relative path or just a file name, it will be placed in *logging.path*. If *logging.file* is not present, the log file is named *spring.log*

### Changing the Log Level

Spring Boot provides a way to modify the log level with properties set in the *application.properties*.

They are prefixed by *logging.level* and the value is the log level, one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF. To set the log level for the root logger use

logging.level.root=WARN

and if you want classes from the Spring Framework to only report errors

logging.level.org.springframework=ERROR

### Custom Logging

The previous setup works for many cases but sometimes your infrastructure is a bit more complex, and you need to have more control in setting up the logger. Spring Boot has you covered too for that.

When it detects a configuration file of your logging system, i.e. *logback.xml* for logback it will use that for initialization. However, Spring will not control your logging system now and might not be able to use certain Spring features, i.e. using Spring profiles in the log configuration.

To enable better Spring integration, it will also look for a configuration file with *-spring* in the name, so *logback-spring.xml* for Logback.

I included one in the Commentstore App which sets up a bit more; and also uses custom MDC variables.

### Follow Requests in Your Landscape

When you are running a bunch of microservices, it really helps when you can track a specific user action throughout your whole system.

When a user action enters the system, it gets a unique ID assigned, and the request is logged. Now you only need to pass the UUID of this request along with the system. To make that as transparent as possible we will use the Mapped Diagnostic Context (MDC) of our logging library.

An MDC provides a way to add user specific data to the current thread so the logging system can use this; it has the benefit that we do not need to pass it along with our call hierarchy by ourselves.

To show you the concept I added an implementation for a ServletFilter in the logging module; class *RequestContextLoggingFilter*. In the sample, the UUID is passed in an HTTP header along the system.

@Override

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

throws IOException, ServletException {

try {

String requestUUID = ((HttpServletRequest) request)

.getHeader(SBBLoggingConstants.REQUEST\_UUID\_HEADER);

if (StringUtils.isEmpty(requestUUID)) {

requestUUID = createId();

LOGGER.info("Got request without {} and assign new {}",

SBBLoggingConstants.REQUEST\_UUID\_HEADER, requestUUID);

MDC.put(SBBLoggingConstants.REQUEST\_UUID\_HEADER,

requestUUID);

}

else {

MDC.put(SBBLoggingConstants.REQUEST\_UUID\_HEADER,

requestUUID);

}

chain.doFilter(request, response);

}

finally {

MDC.clear();

}

}

The *RequestContextLoggingFilter* will check if our header parameter *SBBRequestUUID* is available and if so it will set it as our custom field in the MDC. If no UUID did exists we create a new.

The property, *SBBRequestUUID*, we set here on the parameter can be used in the log pattern, and Logback will automatically add it to the log output.

<encoder>

<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{36} %X{SBBRequestUUID} - %msg%n</pattern>

</encoder>

To use the filter, we must declare it in our *CommentStoreApp* Spring Boot configuration.

@Bean

public Filter initRequestContextLoggingFilter() {

return new RequestContextLoggingFilter();

}

Spring Boot will now add our Filter to the default *DispatcherServlet* and thus running it for all endpoints.

When you run the sample app and test client you will see log entries like:

08:50:11.366 [http-nio-8080-exec-6] INFO d.c.s.c.restapi.ReadController num1 - get comments for pageId 0815

08:50:11.380 [http-nio-8080-exec-6] INFO d.c.s.c.restapi.ReadController num1 - get comments for pageId 0815 - done

*num1* is the UUID used in a manual test.

### Changing the Log Level during Runtime

With Spring Boot 1.5.1 and using the Spring Boot Actuator, we can finally change the log level during runtime. How to set up the Actuator is covered in the next chapter, here we are exploring the feature only.

Under *loggers* the Spring Boot Actuator provides an endpoint to view all configured loggers and their states. The requests we are using here are also included in the Postman collection in the source code.

When you make a GET request to *loggers* you will receive a JSON like:

{

"levels": [

"OFF",

"ERROR",

"WARN",

"INFO",

"DEBUG",

"TRACE"

],

"loggers": {

"ROOT": {

"configured\_level": "INFO",

"effective\_level": "INFO"

},

"de": {

"configured\_level": null,

"effective\_level": "INFO"

},

}

First, we get a list of all supported log levels and second, each logger with its current log level.

To change the level of a logger now, we must make a POST Request to the specific Logger with a JSON containing the new log level.

Let's modify the ROOT Logger by POSTing the following JSON to the logger's endpoint *loggers/ROOT*.

{

"configuredLevel": "DEBUG"

}

In the console log output, you will notice that Spring is now generating much noise with all its DEBUG messages. To change it back, just post:

{

"configuredLevel": "INFO"

}

HINT: When hunting down bugs in a production environment, start with changing the most specific loggers you have configured and from that go up to more generic ones. Changing the ROOT logger or the logger of the major frameworks to DEBUG or even TRACE will flood your disk faster than you can change it back.

### Recap

Before we continue, let's review what we have covered and check your understanding with a short quiz.

* Can you only use SLF4J for logging?
* Where and how do I configure Logging to a file?
* How do I change the log level?
* Can you customize the logging in Spring Boot beyond the previous questions?
* What problem solves the "Follow Requests in your Landscape" section?

In the next chapter, we are going to take a look at metrics and health checks.

@@@@@@@@@@@@@@@@@@@@@@@@@@@

## HealthCheck and Metrics

### Overview

Spring Boot includes several features to help you monitor and manage your application when it is in production. You can manage and monitor your application using HTTP endpoints, JMX or even by a remote shell (SSH or Telnet). The HTTP endpoints are only available with a Spring MVC-based application like ours.

To enable the functionality, add the Actuator Starter to your pom.

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

</dependencies>

The endpoints are configured automatically, and with sensitive defaults, i.e. the shutdown endpoint is disabled.

I will cover the two most valuable endpoints when starting out. For a full list of endpoints, check the Spring Boot documentation.

### Health Check

This endpoint provides information about the health status of the app. It is often used by monitoring software to check if the system is still operating and notifying when the system goes down.

The kind of information depends on your project. In general, Spring Boot checks the application context for all instances of a *HealthIndicator* and makes them available to this endpoint. Common checks like disk space or the status of database connections are added automatically when a starter is used. If something is not covered by Spring Boot, you can always implement your own *HealthIndicator*, and it will show up.

You can access it via HTTP with the */health* endpoint, and you will receive a JSON like

{

"status": "UP",

"diskSpace": {

"status": "UP",

"total": 120007426048,

"free": 59710062592,

"threshold": 10485760

},

"db": {

"status": "UP",

"database": "H2",

"hello": 1

}

}

When security is turned on, and you are not logged in it will only show the status field and hide all details. Even when your whole application is running with security turned on you can disable it for this endpoint alone by adding `endpoints.health.sensitive=false to your *application.properties*.

#### Metrics

The metrics give insight into various stats of your application, like how much memory does the app run with, how much is free, how the heap size is used, how many threads are running and so on. Each starter in the Spring Boot world might add its own metrics to the list. For example, the stats about the data source are only present when a database was detected and configured.

You can access it via HTTP with the */metrics* endpoint, and you will receive a JSON like

{

"mem": 398017,

"mem.free": 143098,

"processors": 4,

"instance.uptime": 6638493,

"uptime": 6651208,

"systemload.average": 1.87353515625,

"heap.committed": 319488,

"heap.init": 65536,

"heap.used": 176389,

"heap": 932352,

"nonheap.committed": 79744,

"nonheap.init": 2496,

"nonheap.used": 78529,

"nonheap": 0,

"threads.peak": 21,

"threads.daemon": 19,

"threads.totalStarted": 25,

"threads": 21,

"classes": 10605,

"classes.loaded": 10606,

"classes.unloaded": 1,

"gc.ps\_scavenge.count": 12,

"gc.ps\_scavenge.time": 234,

"gc.ps\_marksweep.count": 2,

"gc.ps\_marksweep.time": 174,

"httpsessions.max": -1,

"httpsessions.active": 0,

"datasource.primary.active": 0,

"datasource.primary.usage": 0.0,

"gauge.response.health": 124.0,

"gauge.response.metrics": 20.0,

"gauge.response.upload": 212.0,

"gauge.response.unmapped": 63.0,

"gauge.response.id": 76.0,

"gauge.response.list.id": 67.0,

"counter.status.404.list.id": 1,

"counter.status.200.health": 2,

"counter.status.200.list.id": 1,

"counter.status.401.unmapped": 3,

"counter.status.200.upload": 1,

"counter.status.200.metrics": 1,

"counter.status.200.id": 2,

"counter.status.404.id": 1

}

Like with the health stats you can, of course, add your own metrics with *CounterService* and *GaugeService* or by using Dropwizard metrics.

##### Implementing a Custom Counter

To create our own counter we need to inject the *CounterService* in our bean and can use the *increment* and *decrement* methods. The counter lives as long as the Spring context.

@Autowired

private CounterService counterService;

For example in *ReadController* *getComments* I increment the counter every time the method runs.

counterService.increment("commentstore.list\_comments");

When our counter *commentstore.list\_comments* has a value, it is also returned in the metrics request.

### Recap

Before we continue, let's review what we have covered and check your understanding with a short quiz.

* How do you enable the production ready features of Spring Boot?
* Can you secure the endpoints?
* How do you access the health checks?
* Which Spring Bean do you need to use for your own metrics counter?
* Where can you request your metric?

In the next chapter, we are going to take a look at deploying the application.

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

## Deployment of Our Microservice

### Overview

For deploying your Spring Boot application, you have a couple of options.

1. Standalone plain Java application - One single Jar
2. Standalone executable on Unix
3. Deployment as a WAR file
4. Cloud, i.e. Heroku, Cloud Foundry

Remember, in its bare essence; it is a simple Java application. For this book, I cover the most used version and deploy our application to a Unix system.

### Build an Executable Jar

To create an executable jar add the following plugin configuration into your pom:

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<executable>true</executable>

</configuration>

</plugin>

Next time you run mvn package or mvn install, Spring Boot will build a single JAR file with all the dependencies and wrap it with a shell script, so it runs as an executable on Unix systems. It is named *comment-store-1.0.0-SNAPSHOT.jar* and you find it in the *target* directory.

When deploying your application later, you probably do not want the version in the filename or maybe have a completely different one. You can archive this by adding the *fileName* property to the configuration.

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<executable>true</executable>

<finalName>sbb-comment-app</finalName>

</configuration>

</plugin>

Building the package now, the Spring Boot Maven Plugin will name our file *sbb-comment-app.jar*.

The generated app works out of the box with init.d or systemd and accepts *start, stop, restart* and *status* commands.

Copy your application to */var/sbb-comment-app* on your server. Create a symbolic link in *init.d* for your app.

sudo ln -s /var/sbb-comment-app/sbb-comment-app.jar /etc/init.d/sbb-comment-app

Start the app with:

service sbb-comment-app start

Voila!

If it does not start check the log files and if you have not overridden the location you can find it in /var/log/<appname>.log.

Of course, do not forget to secure access to your files and folders when going into production.

You can customize the default script in a certain way, i.e. setting the JAVA\_OPTS. Place the options in a configuration file alongside your jar file. It must use the same application name as your jar file and have *conf* as the file extension.

JAVA\_OPTS=-Xmx1024M

LOG\_FOLDER=/custom/log/folder

### Use Your Own Launch Script for the Unix Executable

Sometimes, your use case is not covered by the default start script, so you have to declare your own.

Add the property *embeddedLaunchScript* to the Spring Boot plugin declaration in your pom and specify the name of the launch script. Your script is a regular one which should run on your server platform.

As an example, I included the default Spring Boot script in the code and enhanced it to included options set in a different file. We had this use cases in deploying our production application to distinguish between parameters set by us and parameters set by our managed hosting service. The script will look up additional environment configuration in a file named *spring\_<app-name>* in */etc/default/*.

### Recap

Let's review what we have covered and check your understanding with a short quiz.

* Does Spring Boot create a Unix executable of your application by default?
* Can you start a Spring Boot application as a simple Java program?
* Can the Unix executable be used with init.d?
* How can you define properties, like JAVA\_OPTS for your application?
* How can you define your own start script when deploying as a Unix executable?