# Spring in 15 minutes

by David Kiss (http://kaviddiss.com)

# Table of Contents

## Welcome to the Spring in 15 Minutes tutorial

This step-by-step tutorial was created to help you get started with Spring and take you through creating a sample blog application using Spring framework v4, Spring Boot, Spring Data and Spring Security frameworks.

If you find this document useful, you'll find more Spring tutorials at http://kaviddiss.com/learn-spring/.

In case you have a question on Spring, found an error in the tutorial, or just want to compliment on the nice fonts in this document, leave a message at http://kaviddiss.com/contact/.

## Prerequisites

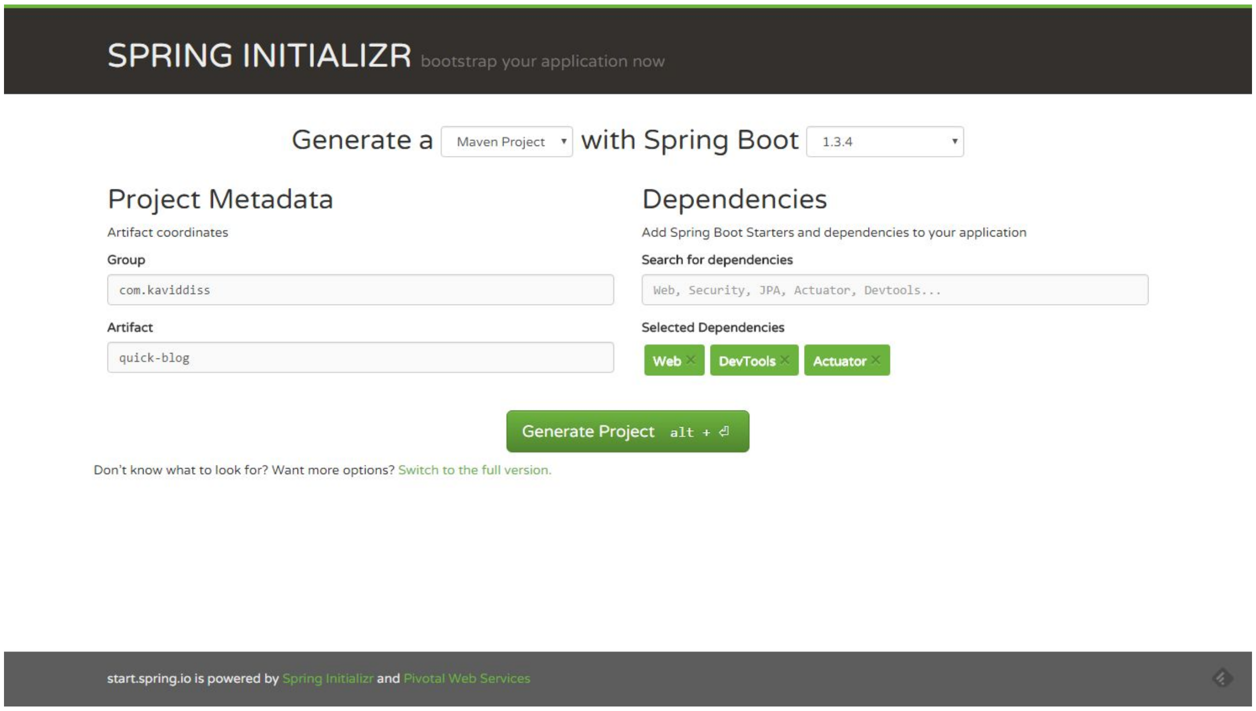
- Previous experience with Java
- An IDE to editing Java source code. I personally prefer using IntelliJ (https://www.jetbrains.com/idea/download), but feel free to use Eclipse (http://www.eclipse.org/downloads/), Netbeans (https://netbeans.org/downloads/) or anything else you feel comfortable with.
- JDK installed and configured for the project in your IDE, preferably v1.8, the latest, version

## Step #1 - Create your first project

In this section we'll create a very basic Java project that we'll use for creating the blog application.

To get started, navigate to the Spring Initializr webpage at http://start.spring.io in your favourite browser. On this page you'll be able to generate a basic Spring framework / Spring Boot based project.

> **What is Spring Boot?**
> In case you're not familiar with Spring Boot, it's one of the latest additions to the many existing Spring frameworks. It was inspired by Dropwizard (http://www.dropwizard.io/) and it's main goal is to simplify working with Spring and supporting development of microservices using Spring.


SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Spring Boot 1.3.4

Project Metadata

Artifact coordinates

Group: com.kaviddiss

Artifact: quick-blog

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies: Web, Security, JPA, Actuator, Devtools...

Selected Dependencies: Web, DevTools, Actuator

Don't know what to look for? Want more options? Switch to the full version.

start.spring.io is powered by Spring Initializr and Pivotal Web Services

On the Spring Initializr webpage select Maven Project and the latest stable Spring Boot version (currently 1.3.5).

> **What is Maven?**
> Apache Maven (http://maven.apache.org/) is probably the most commonly used open-source build and dependency management tool for Java. It is similar to NPM in Node.js or RubyGems in Ruby.

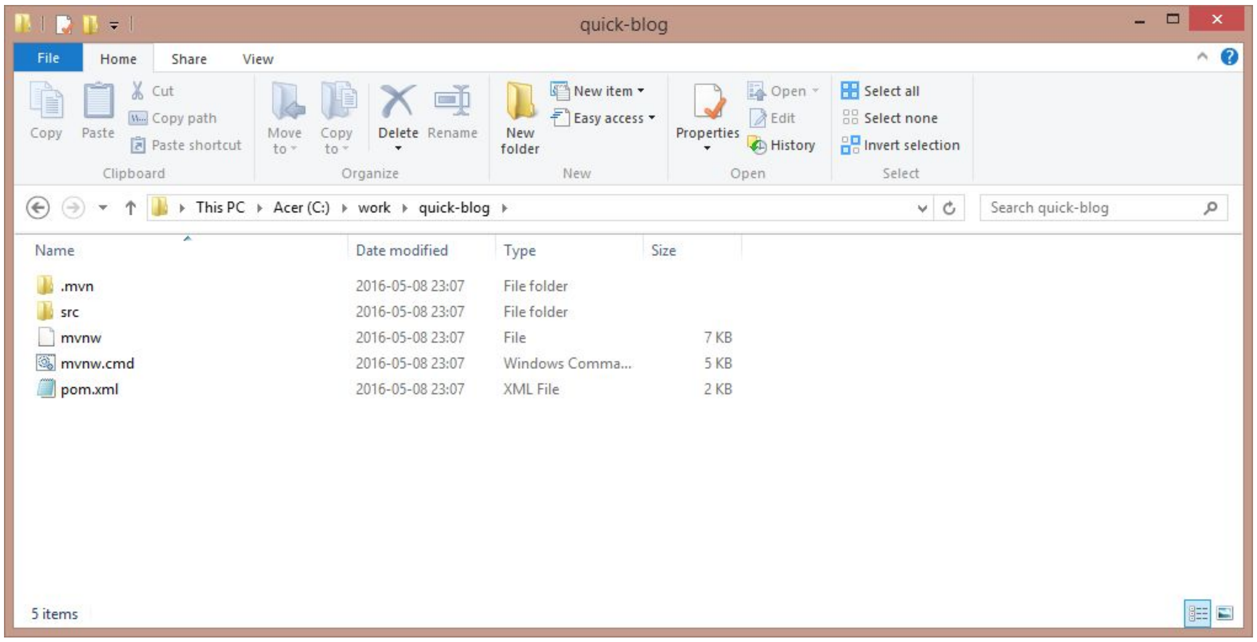
The Group field allows grouping build artifacts together. For example, if your company's website is http://mycompany.com, you'd use com.mycompany as the Group for all the java projects within the company.

The Artifact field along with the Group field allows to uniquely identify a Java project in Maven. It's generally a good idea to set it to a meaningful value that describes your project.

In the Dependencies field select these options:

- *Web:* allows us to create web applications
- *DevTools:* a library used locally in development mode that can automatically restart your Spring web application whenever files in the classpath change (in IntelliJ it can be triggered by clicking Build / Make Project menu item, in Eclipse by saving a modified file)
- *Actuator:* a library that provides useful information (monitoring, health, etc.) on the running Spring application

At this point you can click the Generate Project button that will generate a zip file that you'll need to download and extract on your computer.



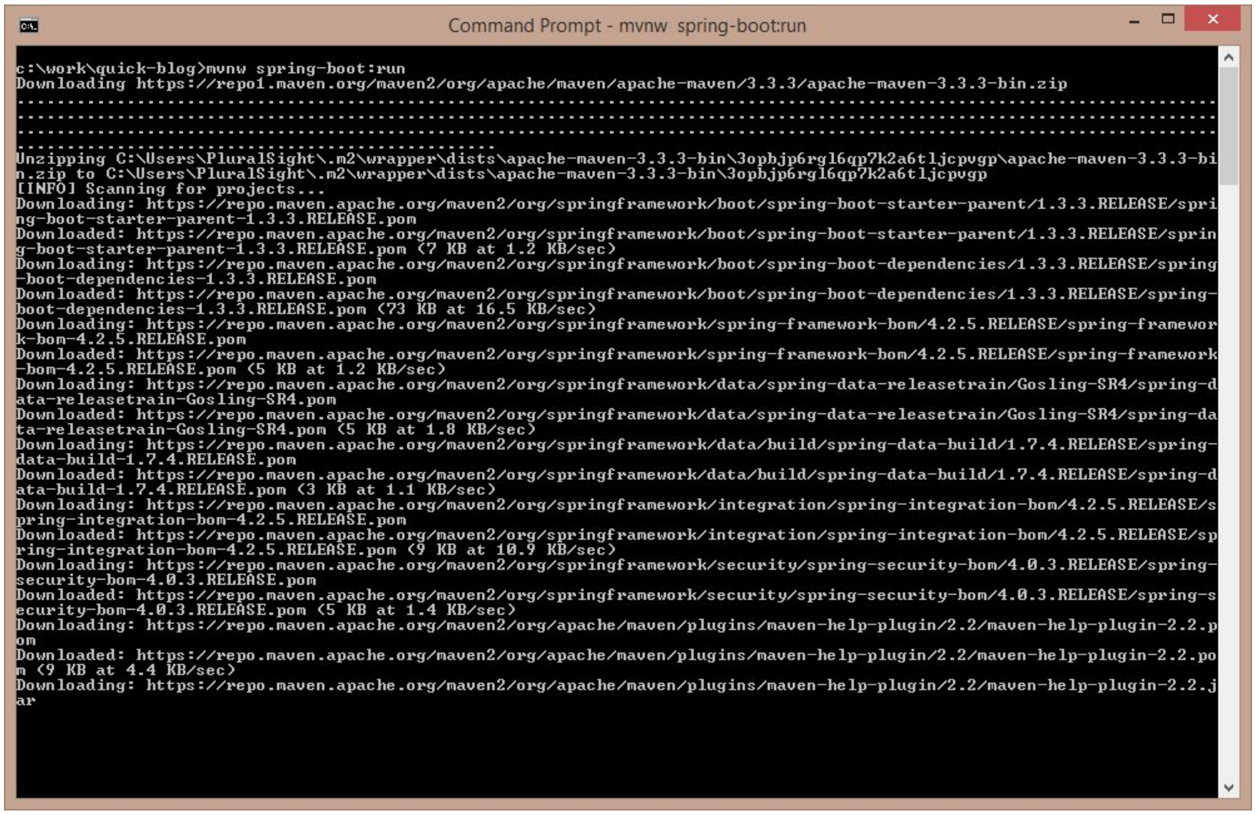
| Name | Date modified | Type | Size |
|---|---|---|---|
| .mvn | 2016-05-08 23:07 | File folder | |
| src | 2016-05-08 23:07 | File folder | |
| mvnw | 2016-05-08 23:07 | File | 7 KB |
| mvnw.cmd | 2016-05-08 23:07 | Windows Comma... | 5 KB |
| pom.xml | 2016-05-08 23:07 | XML File | 2 KB |

Starting from the bottom,

- `pom.xml:` is the Maven project descriptor file where you can configure the project's dependencies and build steps
- `mvnw` or `mvnw.cmd`*:* if you don't have Maven installed yet, running these commands will install it locally
- `src`*:* folder containing all source files needed to build this project
- *`.mvn:`* folder needed for `mvnw` command

- `~/.m2/repository`: this folder is not inside the project folder, rather it lives on it’s own and includes a local cache of all the artifacts used as a dependency for your local Java projects

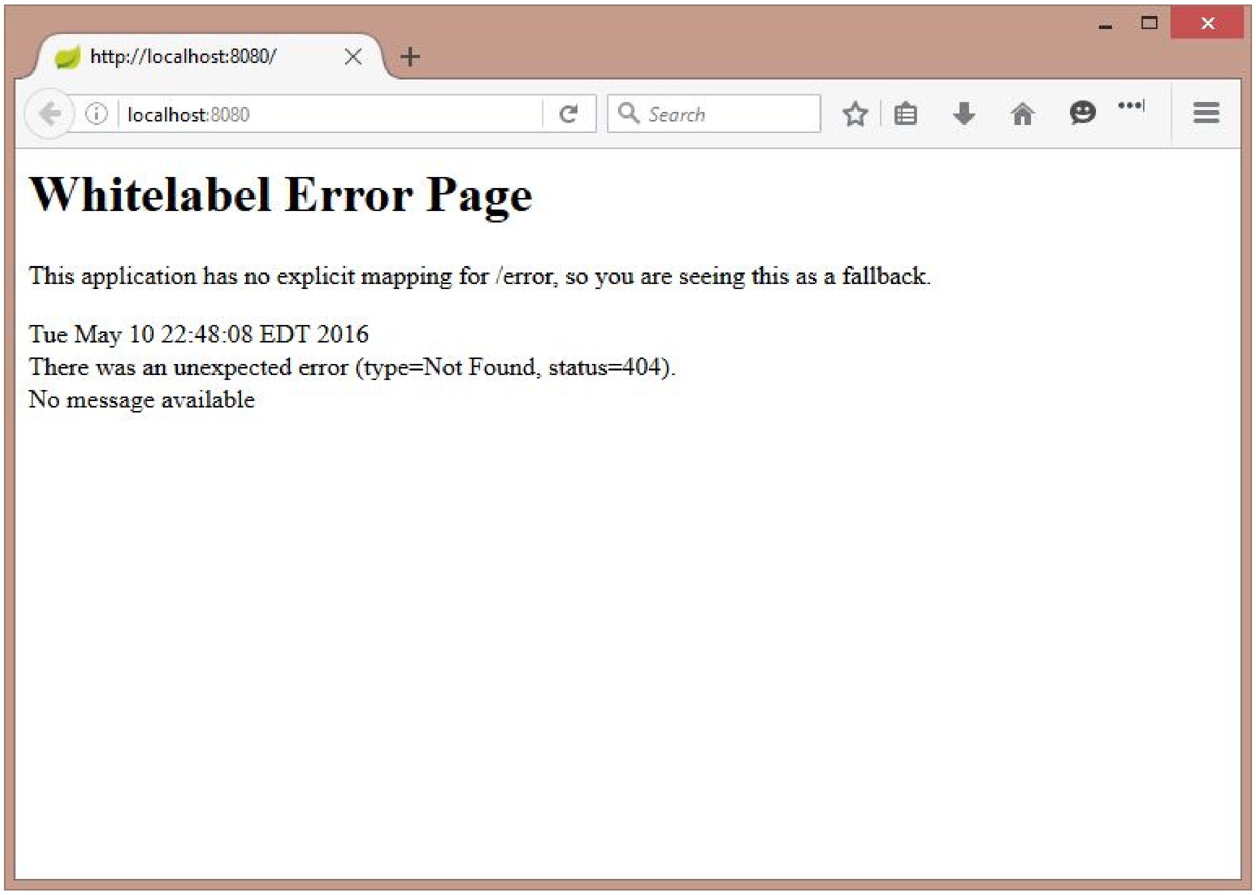
Now in the command line run: `mvnw spring-boot:run`



```
c:\work\quick-blog>mvnw spring-boot:run
Downloading https://repo1.maven.org/maven2/org/apache/maven/apache-maven/3.3.3/apache-maven-3.3.3-bin.zip
........................................................................................................................
........................................................................................................................
........................................................................................................................
................................................
Unzipping C:\Users\PluralSight\.m2\wrapper\dists\apache-maven-3.3.3-bin\3opbjp6rgl6qp7k2a6tljcpvgp\apache-maven-3.3.3-bi
n.zip to C:\Users\PluralSight\.m2\wrapper\dists\apache-maven-3.3.3-bin\3opbjp6rgl6qp7k2a6tljcpvgp
[INFO] Scanning for projects...
Downloading: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-starter-parent/1.3.3.RELEASE/spri
ng-boot-starter-parent-1.3.3.RELEASE.pom
Downloaded: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-starter-parent/1.3.3.RELEASE/sprin
g-boot-starter-parent-1.3.3.RELEASE.pom (7 KB at 1.2 KB/sec)
Downloading: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-dependencies/1.3.3.RELEASE/spring
-boot-dependencies-1.3.3.RELEASE.pom
Downloaded: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-dependencies/1.3.3.RELEASE/spring-
boot-dependencies-1.3.3.RELEASE.pom (73 KB at 16.5 KB/sec)
Downloading: https://repo.maven.apache.org/maven2/org/springframework/spring-framework-bom/4.2.5.RELEASE/spring-framewor
k-bom-4.2.5.RELEASE.pom
Downloaded: https://repo.maven.apache.org/maven2/org/springframework/spring-framework-bom/4.2.5.RELEASE/spring-framework
-bom-4.2.5.RELEASE.pom (5 KB at 1.2 KB/sec)
Downloading: https://repo.maven.apache.org/maven2/org/springframework/data/spring-data-releasetrain/Gosling-SR4/spring-d
ata-releasetrain-Gosling-SR4.pom
Downloaded: https://repo.maven.apache.org/maven2/org/springframework/data/spring-data-releasetrain/Gosling-SR4/spring-da
ta-releasetrain-Gosling-SR4.pom (5 KB at 1.8 KB/sec)
Downloading: https://repo.maven.apache.org/maven2/org/springframework/data/build/spring-data-build/1.7.4.RELEASE/spring-
data-build-1.7.4.RELEASE.pom
Downloaded: https://repo.maven.apache.org/maven2/org/springframework/data/build/spring-data-build/1.7.4.RELEASE/spring-d
ata-build-1.7.4.RELEASE.pom (3 KB at 1.1 KB/sec)
Downloading: https://repo.maven.apache.org/maven2/org/springframework/integration/spring-integration-bom/4.2.5.RELEASE/s
pring-integration-bom-4.2.5.RELEASE.pom
Downloaded: https://repo.maven.apache.org/maven2/org/springframework/integration/spring-integration-bom/4.2.5.RELEASE/sp
ring-integration-bom-4.2.5.RELEASE.pom (9 KB at 10.9 KB/sec)
Downloading: https://repo.maven.apache.org/maven2/org/springframework/security/spring-security-bom/4.0.3.RELEASE/spring-
security-bom-4.0.3.RELEASE.pom
Downloaded: https://repo.maven.apache.org/maven2/org/springframework/security/spring-security-bom/4.0.3.RELEASE/spring-s
ecurity-bom-4.0.3.RELEASE.pom (5 KB at 1.4 KB/sec)
Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-help-plugin/2.2/maven-help-plugin-2.2.p
om
Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-help-plugin/2.2/maven-help-plugin-2.2.po
m (9 KB at 4.4 KB/sec)
Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-help-plugin/2.2/maven-help-plugin-2.2.j
ar
```

This command will run the empty Spring Boot application and when you execute the command for the first time, it will also download Maven and the project’s dependencies (into the `~/.m2/repository` folder).

If you open your browser and go to [http://localhost:8080](http://localhost:8080) now, you should see something like this:



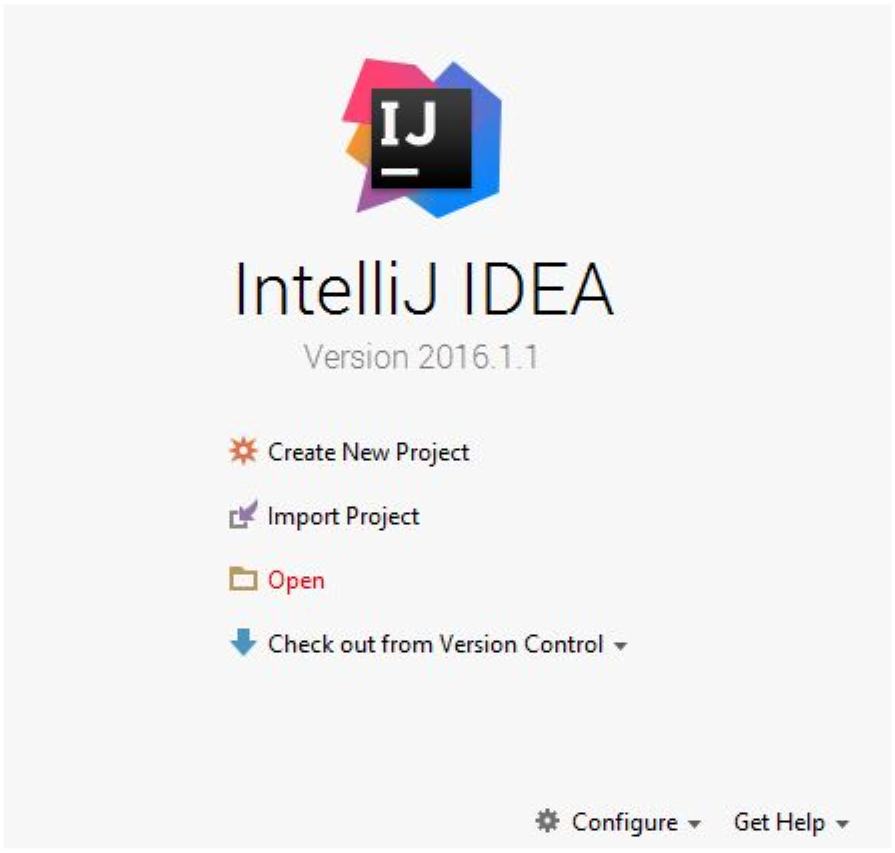
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

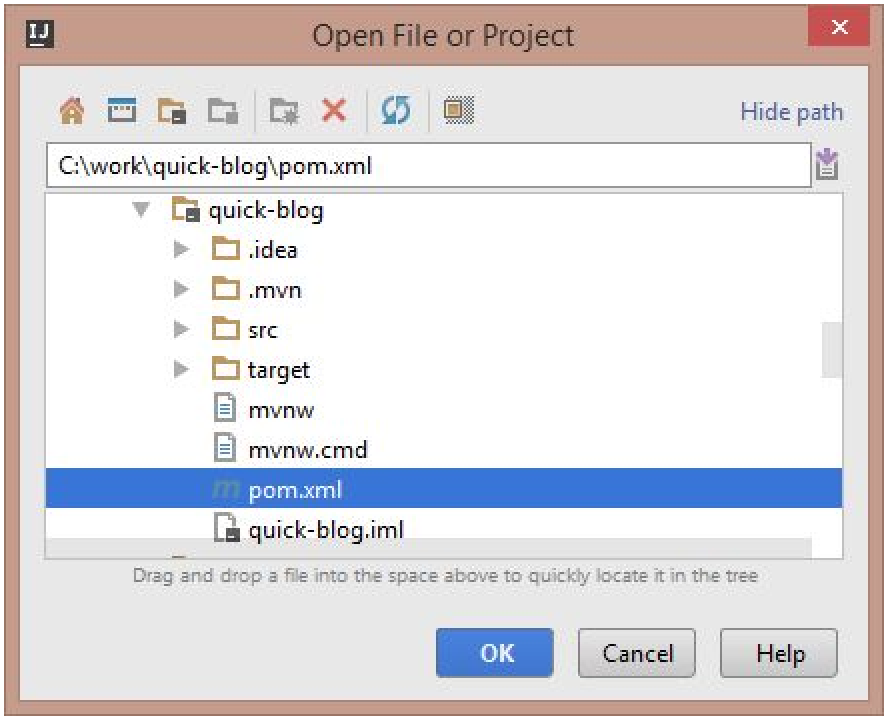
Tue May 10 22:48:08 EDT 2016
There was an unexpected error (type=Not Found, status=404).
No message available

It's not the most beautiful welcome page, right? We're faced with this HTTP 404 (Not Found) error message since our application doesn't have any web pages or REST services configured yet. At least we know the application is running!

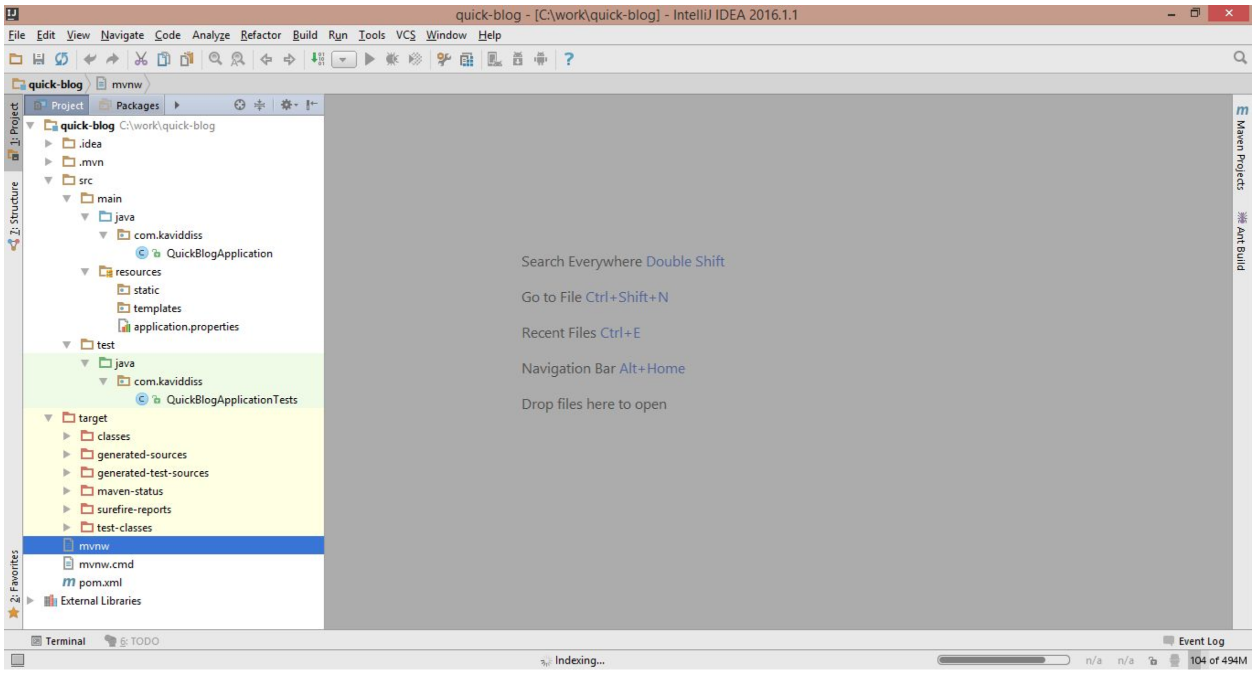
Now let's open the project in IntelliJ (or your favourite IDE). Start IntelliJ and click Open (or File / Open…menu, if you have another project already open).



Then select the `pom.xml` file in your project and click OK:



You should see something like this in IntelliJ:



```
quick-blog C:\work\quick-blog
  .idea
  .mvn
  src
    main
      java
        com.kaviddiss
          QuickBlogApplication
      resources
        static
        templates
        application.properties
    test
      java
        com.kaviddiss
          QuickBlogApplicationTests
  target
    classes
    generated-sources
    generated-test-sources
    maven-status
    surefire-reports
    test-classes
  mvnw
  mvnw.cmd
  pom.xml
External Libraries
```

## Step #2 - Create static content

Now we have the project created, it's time to add some static content to the web application.

Let's fix the HTTP 404 issue by returning a hard-coded "My Quick Blog" when the user navigates to http://localhost:8080/ in the browser.

To be able to do that, we'll create below MainController class in the `com.kaviddiss.web` package:

```java
package com.kaviddiss.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MainController {
   @RequestMapping("/")
   @ResponseBody
   public String index() {
       return "My Quick Blog";
   }
}
```

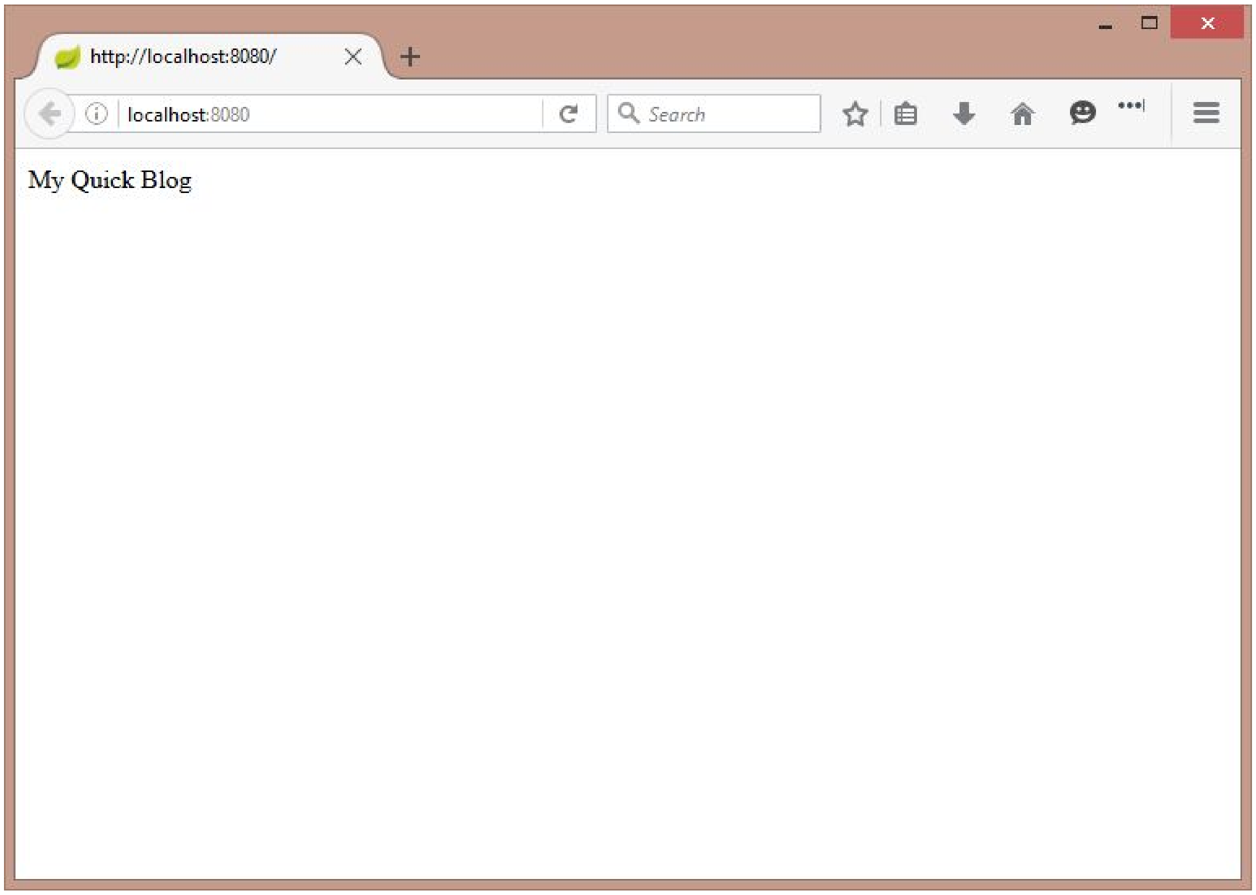
The MainController is annotated with the `@Controller` annotation which tells Spring that this class is a Spring component (Spring Bean) and it implements the **C**ontroller from the MVC (**M**odel-**V**iew-**C**ontroller) pattern.

If you're wondering what that means, let's take a look at the `index()` method.

Public methods marked with `@RequestMapping` inside controller classes will map to a context path (for example: /product/12345). In our case, the `index()` method is mapped to the root path ("/"). This maps to the http://localhost:8080/ url when running the application locally on the 8080 port.

You'll notice that all what the `index()` method does is returning the text "My Quick Blog". Since the method is annotated with `@ResponseBody`, Spring will use the result of the `index()` method as the body of the HTTP response it returns.

Rebuild the project or restart the web application for these changes to take effect:



My Quick Blog

## Step #3 - Create dynamic content

Let’s make the previous page a little nicer by using some HTML code. In addition to that, we’ll make it dynamic: it will take a `name` request param which will be included in the HTML response.

In order to generate a dynamic HTML page, we’ll use a template engine called Thymeleaf to render the content.

> **What is Thymeleaf?**
> Thymeleaf (http://www.thymeleaf.org/) is an open-source and Java-based template engine that is fully integrated with Spring and aims to be a substitute for JSP.

The first step is to add the Thymeleaf dependencies to the project. In the pom.xml file add the thymeleaf dependency under project/dependencies (around line 39):

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

Click the Enable Auto Import link in IntelliJ, in case that notification appears in the top right corner. This will configure IntelliJ to automatically import any new dependencies we add manually to the `pom.xml`.

Create an index.html file under `src/main/resources/templates` with below content:

```
<html>
<body>
   <h1>My Quick Blog</h1>
   <p>
       Welcome to my blog, <span th:text="${name}"></span>!
   </p>
</body>
</html>
```

Notice the `th:text="${name}"` section in the file which instructs Thymeleaf to set the content of the <span> tag to the name parameter passed to Thymeleaf.

We'll configure Spring to return this HTML code when users navigate to the root context path by updating the `index()` method in the MainController class:

```
@RequestMapping("/")
public ModelAndView index(@RequestParam("name") String name) {
   Map<String, Object> model = new HashMap<>();
   model.put("name", name);
   return new ModelAndView("/index", model);
}
```

First, the `@ResponseBody` annotation got removed, then the method now takes a request parameter called name (see the `@RequestParam` annotation`)`.

Also, instead of String, now we return a ModelAndView object which allows us to configure the name of the view to be returned `"/index"` and pass in various parameters to the view (for example: `name`) that can be referenced in the HTML template to make the content dynamic.

Remember, the `"/index"` refers to the view used for rendering and **V**iew is the **V** in the MVC pattern we discussed earlier.

**What is a View?**
View is a representation of the data we want to display to the user. In this case, a view is an HTML file and view name is the filename of the view (without the .html extension)

Earlier in the tutorial we placed the index.html file under the `src/main/resources/templates` folder. If you’re not familiar with Maven, the `src/main/resources` folder is used for resource files / static content that will be copied to the `target/classes` folder along with the Java class files during the build process and therefore will be available on the classpath at runtime.

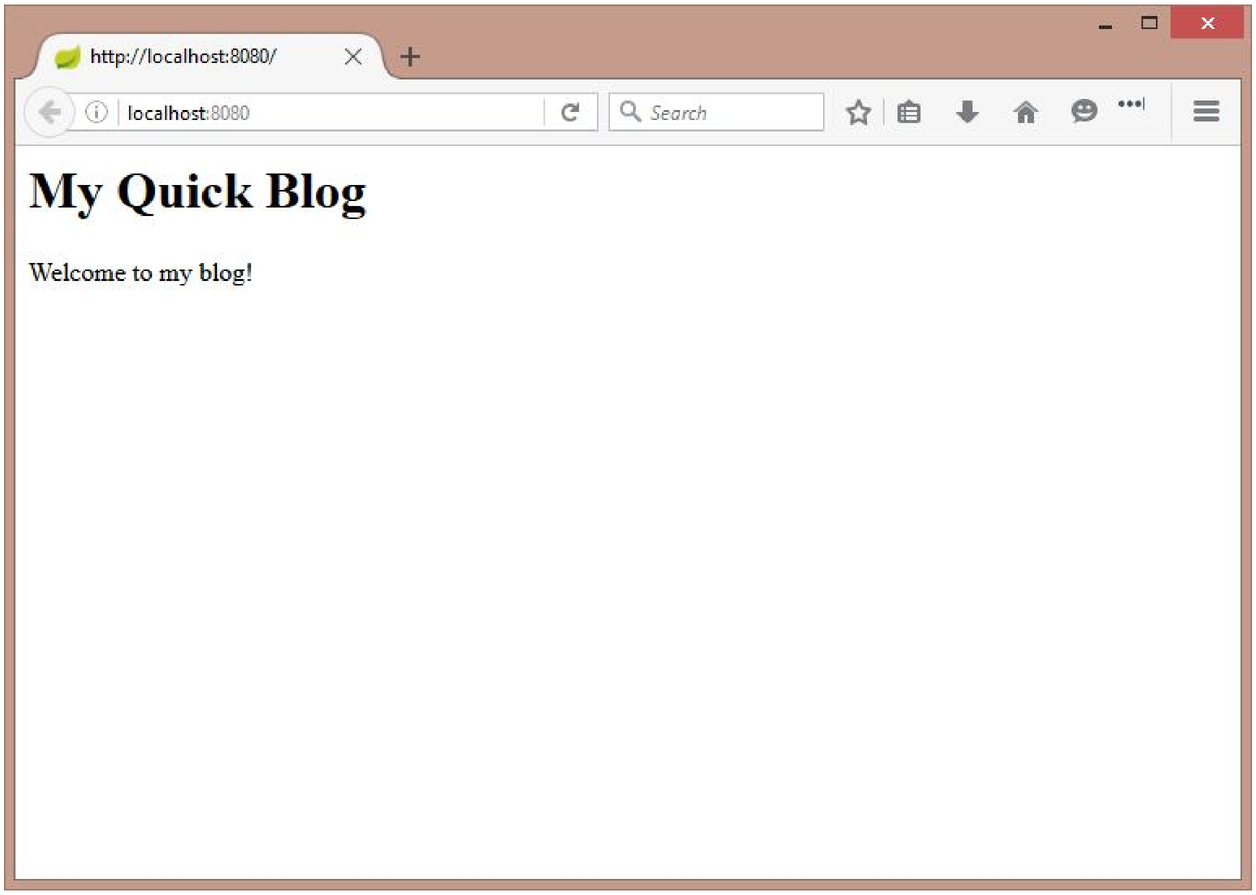
Now the `src/main/resources/templates` is folder used by Spring for storing template files (for rendering HTML, as an example). When the `index()` method returns “/index”, Spring will look for a template file under the `src/main/resources/templates` folder with the relative path of “/index.html” (the .html is the default postfix for view names) that would translate to `src/main/resources/templates/index.html`, which we created earlier.

Once Spring finds this `index.html` file, it will render it using Thymeleaf, the templating engine.

To ensure that changes to HTML files take effect after rebuilding the project without any restarts,, we need to turn off the Thymeleaf caching by updating the `src/main/resources/application.properties` file by adding below line:

```
spring.thymeleaf.cache=false
```

If you followed these steps, let’s rebuild the project to trigger the auto-reload and go to http://localhost:8080/ in the browser to see the magic happen:



# My Quick Blog

Welcome to my blog!

After taking a short glimpse into rendering UI using Spring Boot and Thymeleaf now let's take a look into working with databases.

## Step #4 - Create our data model

In the previous steps we created a controller and a simple HTML view. In this step we're going to create our data **M**odel, the M in the MVC pattern.

Now let's create a Post entity!

> **What is an entity?**
> An entity is a Java object that is mapped to a database table and is used to store and retrieve data from the DB using an ORM (Object Relational Mapping) tool, like Hibernate (http://hibernate.org).

First, let's include the `spring-boot-starter-data-jpa` and `h2` dependencies in the pom.xml file:

```
<!-- DB access -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

> **What is Spring Data JPA?**
> Spring Data JPA is a Spring framework that makes it very easy to work with SQL databases using the JPA (Java Persistence Api) standard
>
> **What is H2?**
> H2 is a Java-based SQL database that can be used in both embedded and standalone scenarios and it's commonly used when developing/testing web applications locally.

Next, let's create the `com.kaviddiss.domain` package under `src/main/java` and create the `Post` class within that package:

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import java.io.Serializable;

@Entity
public class Post {
   @Id
   @GeneratedValue
   private Long id;
```

```
    private String title;
    private String body;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getBody() {
        return body;
    }

    public void setBody(String body) {
        this.body = body;
    }
}
```

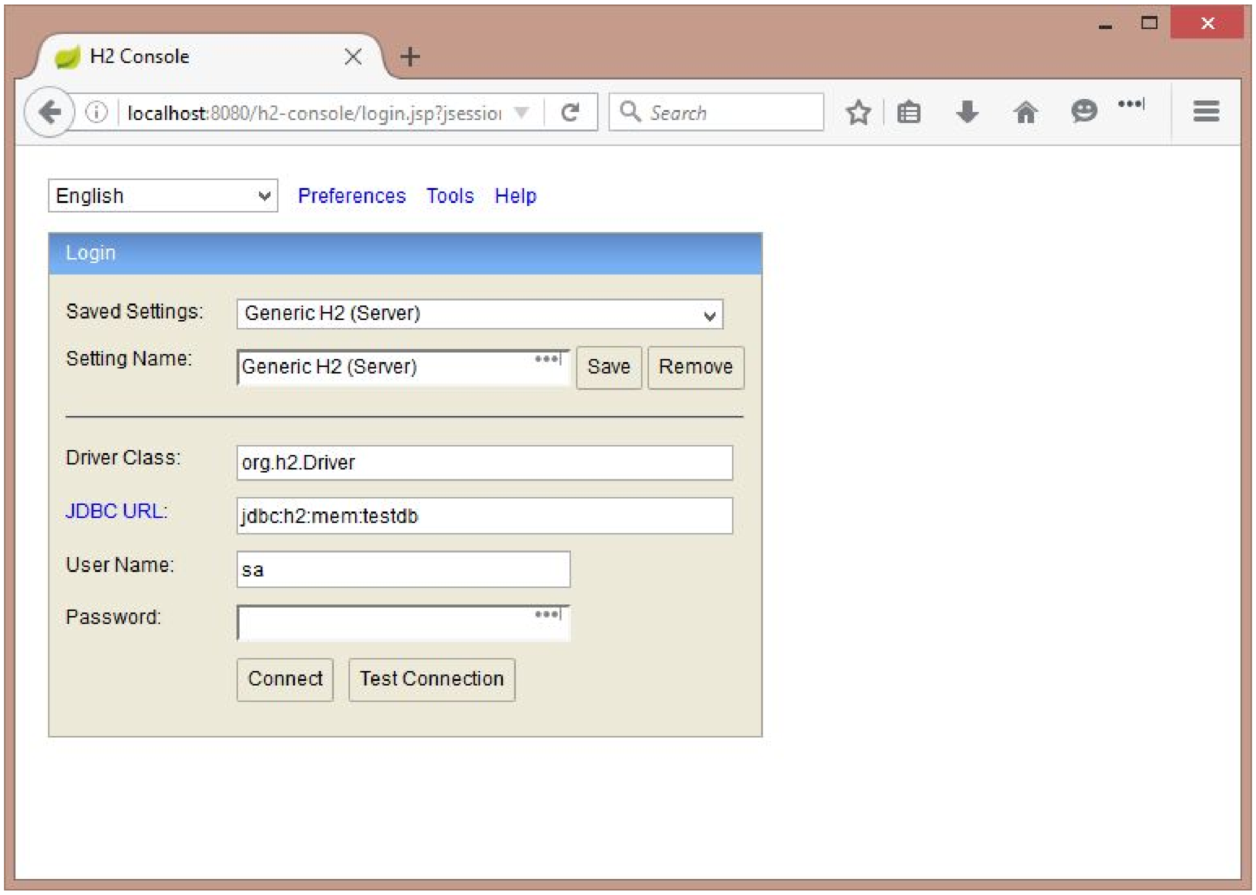
If the `spring-boot-starter-data-jpa` dependency is added to the project, Spring Boot will automatically try to initialize the Spring Data JPA framework. Spring Data JPA then will scan the classpath for classes marked with the `@Entity` annotation (ie: the Post class) and will map them to their corresponding DB tables including the columns defined by the entity fields. By default, it will re-create the tables every time the application starts up.

Now let's look at the fields of the Post entity:

- id - the `@Id` annotation on this field marks it as the primary key for this entity. It also has the `@GeneratedValue` annotation which means the primary key will be automatically generated and will be sequentially increased every time a new entity is created in the DB
- title - this is a String field / varchar column
- body - this is a String field / varchar column

Since the `h2` dependency is added to the classpath, Spring will automatically create a H2 datasource that will be responsible for creating connections to the H2 database when the application tries to store/retrieve data to/from the DB. The default JDBC url for the H2 database is `jdbc:h2:mem:testdb`, the driver class name is `org.h2.Driver`, the username is `sa` and there's no password. In case you're not familiar with JDBC, the JDBC url, driver class name, username and password are the most important and most commonly used DB connection details.

Now restart the application and go to http://localhost:8080/h2-console:



Saved Settings: Generic H2 (Server)

Setting Name: Generic H2 (Server)

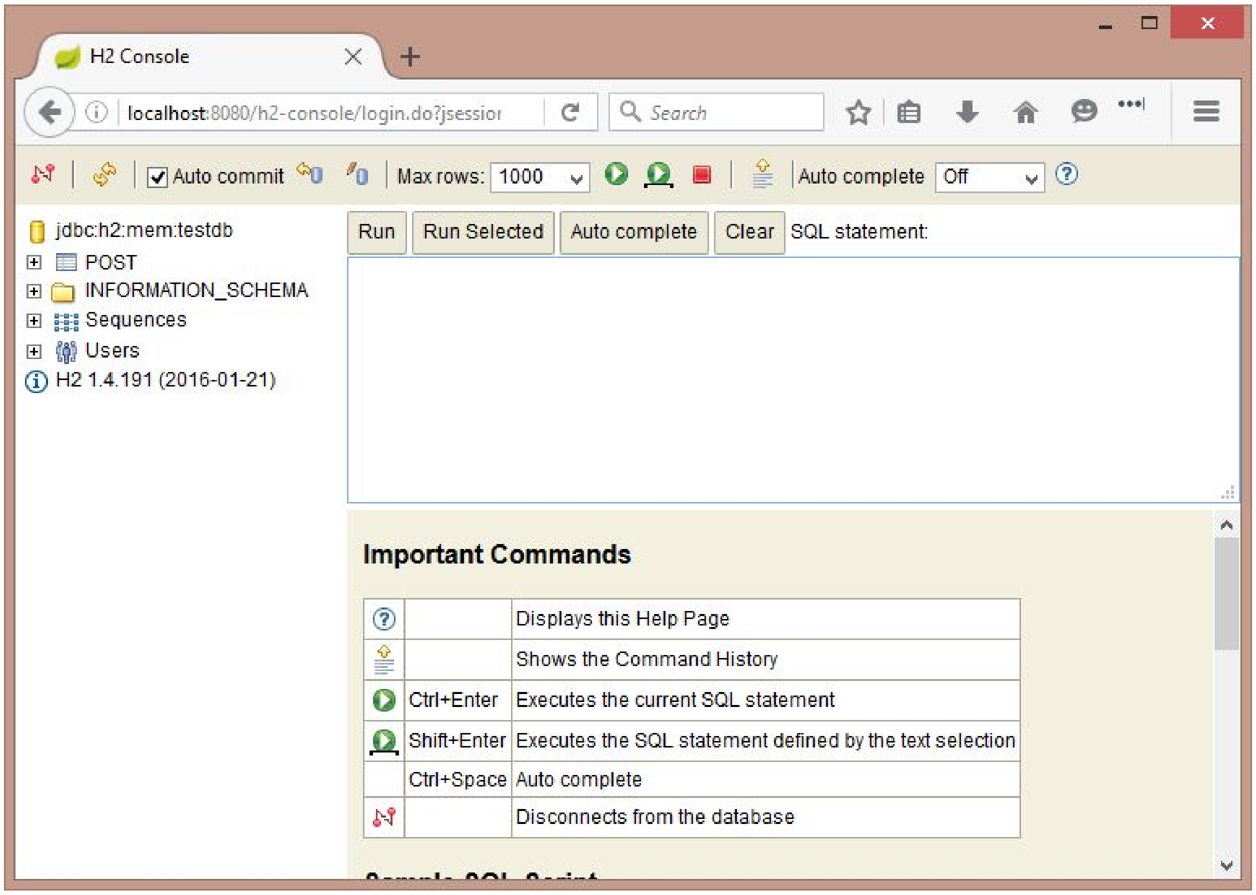
Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:testdb

User Name: sa

Password:

If you click the Connect button, you should see the POST table in the left navigation bar:



Important Commands

| | | |
|---|---|---|
| | | Displays this Help Page |
| | | Shows the Command History |
| | Ctrl+Enter | Executes the current SQL statement |
| | Shift+Enter | Executes the SQL statement defined by the text selection |
| | Ctrl+Space | Auto complete |
| | | Disconnects from the database |

It would be nice to pre-populate the DB with some data for testing so we don't have to create those records manually every time the application is started up.

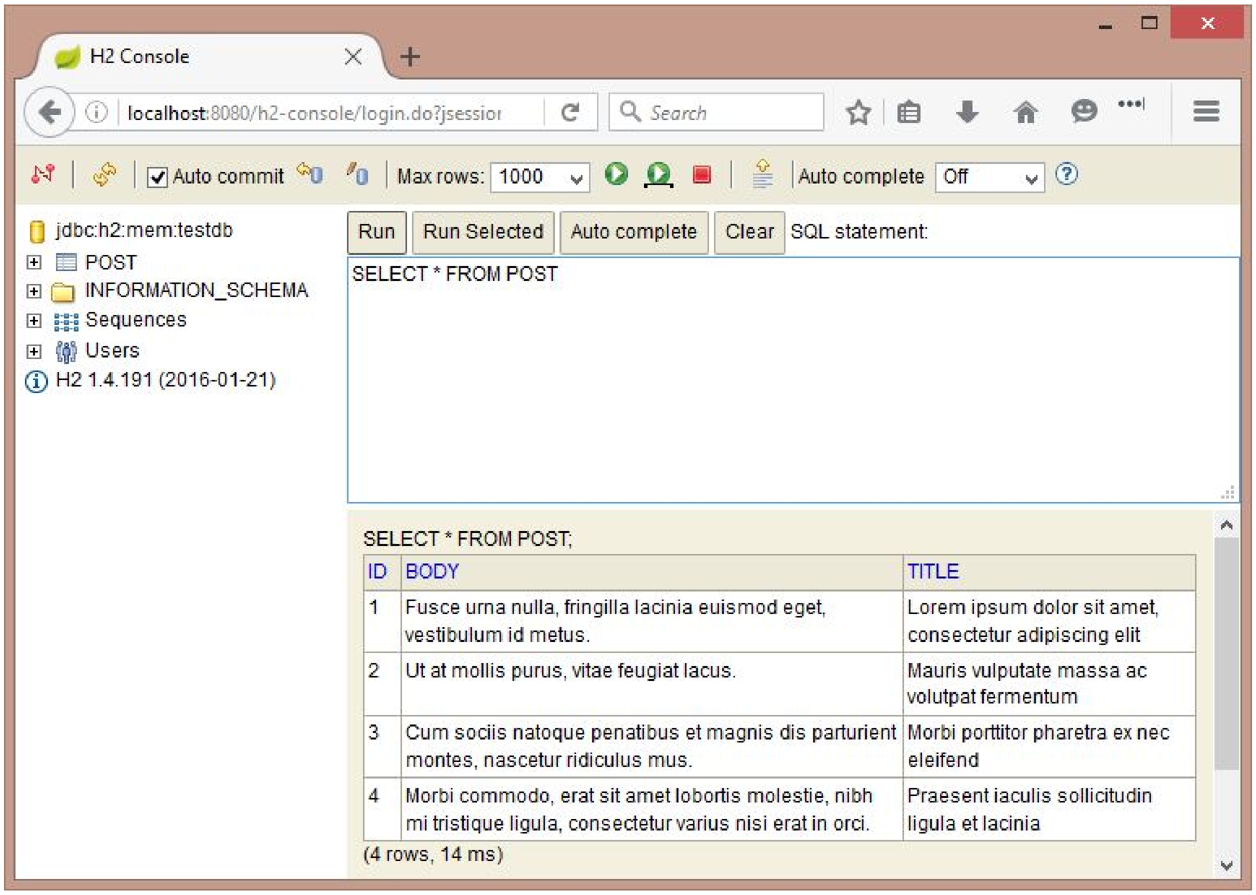
Actually Spring provides a few ways to do that, but for now, let's take a look at one of the most easiest option. Spring Boot provides a simple way to initialize a database by placing a file named import.sql file under `src/main/resources` folder. During application startup time, Spring Boot looks for that file and if it exists, it will execute its content.

With that said, let's create that file with below content that will insert 4 Posts into the database:

```sql
insert into post (title, body) values ('Lorem ipsum dolor sit amet, consectetur
adipiscing elit', 'Fusce urna nulla, fringilla lacinia euismod eget, vestibulum id
metus.');
insert into post (title, body) values ('Mauris vulputate massa ac volutpat fermentum',
'Ut at mollis purus, vitae feugiat lacus.');
insert into post (title, body) values ('Morbi porttitor pharetra ex nec eleifend',
'Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus
mus.');
```

```
insert into post (title, body) values ('Praesent iaculis sollicitudin ligula et
lacinia', 'Morbi commodo, erat sit amet lobortis molestie, nibh mi tristique ligula,
consectetur varius nisi erat in orci.');
```

Let's validate the script was executed by typing SELECT * FROM POST into the H2 Console text area and clicking the Run button:



```
SELECT * FROM POST
```

SELECT * FROM POST;

| ID | BODY | TITLE |
|---|---|---|
| 1 | Fusce urna nulla, fringilla lacinia euismod eget, vestibulum id metus. | Lorem ipsum dolor sit amet, consectetur adipiscing elit |
| 2 | Ut at mollis purus, vitae feugiat lacus. | Mauris vulputate massa ac volutpat fermentum |
| 3 | Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. | Morbi porttitor pharetra ex nec eleifend |
| 4 | Morbi commodo, erat sit amet lobortis molestie, nibh mi tristique ligula, consectetur varius nisi erat in orci. | Praesent iaculis sollicitudin ligula et lacinia |

(4 rows, 14 ms)

# Step #5 - Manipulating the data

Since we have some dummy test data already inserted into our database, we might as well write some queries to fetch them from the DB.

## Using Spring Data JPA

In this section we're going to create two REST services, one that will find a blog post by its title and another one that will search blog posts based on a keyword in the post's body.

The Spring Data JPA framework allows us to create interfaces that define access to the database and it will automatically generate the db queries for the methods we declare within the repository interfaces.

Now, let's create a PostRepository interface under `com.kaviddiss.repository`:

```java
package com.kaviddiss.repository;

import com.kaviddiss.domain.Post;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface PostRepository extends CrudRepository<Post, Long> {
   Post findOneByTitle(String title);

   List<Post> findByBodyContaining(String keyword);
}
```

This interface is annotated with the `@Repository` annotation which tells Spring this is a Spring component and that it's a Spring Data repository interface.

There are two methods in `PostRepository` and they both follow a naming convention that allows Spring Data to understand the meaning of the method names and generate their matching JPA (and in turn SQL) queries.

The first method, `findOneByTitle()`, returns a single Post entity that has a title matching the provided parameter, and the second method, `findByBodyContaining()`, returns a list of posts that include the `keyword` parameter in their body.

See, this is easy. We didn't even write a single SQL query!

Here's a list of some of the supported keywords that can be used in method names: `And, Or, Equals, Between, LessThan, LessThanEqual, GreaterThan, GreaterThanEqual, After, Before, IsNull, NotNull, Like, Containing, OrderBy, In, Not`.

You may notice that the interface also extends `CrudRepository` which comes from the Spring Data framework and it declares many methods around common functionalities for accessing the database: save(), findOne(), findAll(), exists(), count(), delete(), deleteAll(), etc.

Spring Data will also automatically generate JPA (and SQL) queries for methods declared in parent interfaces that our interface extends.

What we'll do here next is to create REST services to expose the methods in the `PostRepository` class. There are multiple ways to do that. First, similar to the MainController class we created earlier we'll create a `PostRestController`.

## Creating a RestController

In this example you'll see how to create REST services manually and also, how to reference other Spring components (aka Beans):

Under the `com.kaviddiss.web` package create the PostRestController class with below code:

```
package com.kaviddiss.web;

import com.kaviddiss.domain.Post;
import com.kaviddiss.repository.PostRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
public class PostRestController {
   private final PostRepository postRepository;

   @Autowired
   public PostRestController(PostRepository postRepository) {
       this.postRepository = postRepository;
   }

   @RequestMapping(value = "/post/findByTitle")
   public Post findOneByTitle(@RequestParam("title") String title) {
       return postRepository.findOneByTitle(title);
   }
```

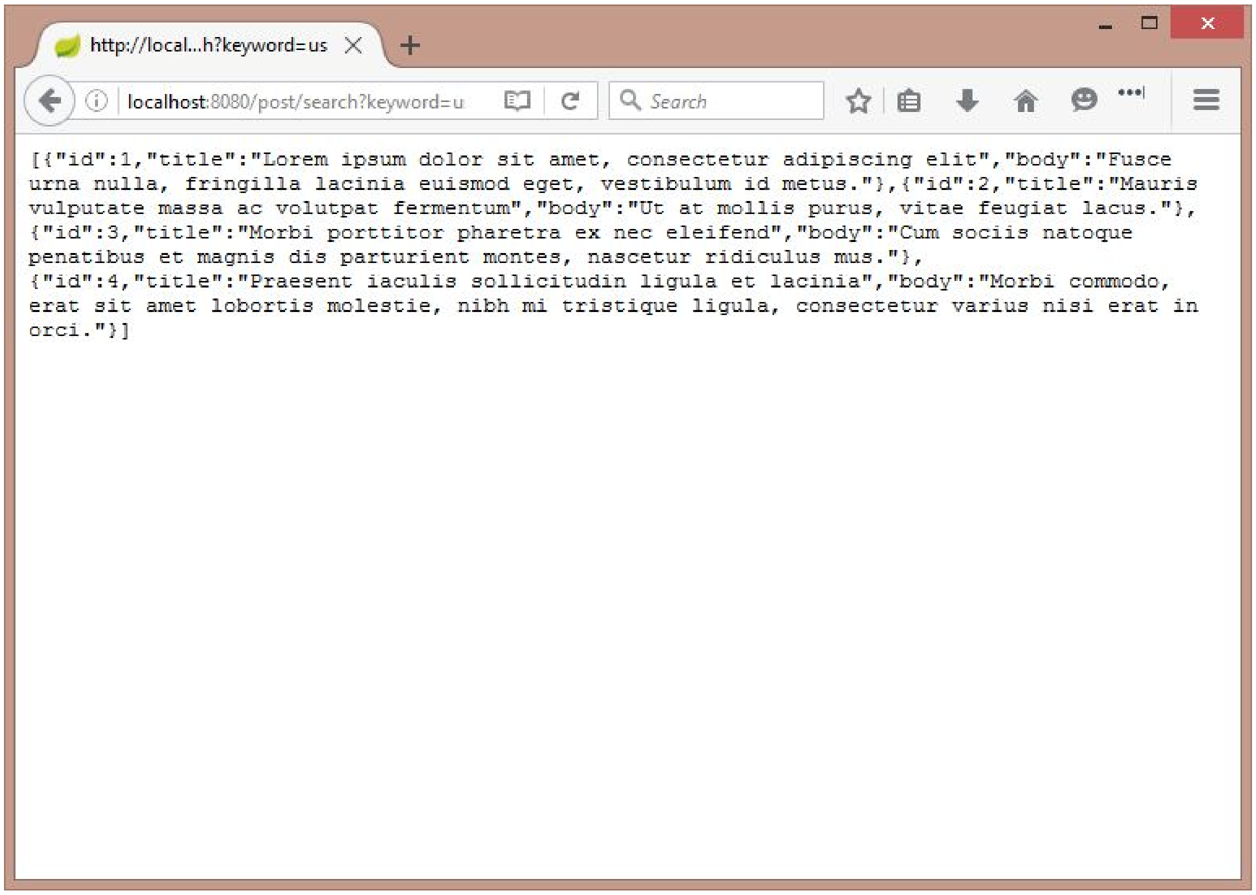
```
    @RequestMapping(value = "/post/search")
    public List<Post> findByBodyContaining(@RequestParam("keyword") String keyword) {
        return postRepository.findByBodyContaining(keyword);
    }

    @RequestMapping(value = "/post/count")
    public long count() {
        return postRepository.count();
    }
}
```

This controller uses the `@RestController` annotation instead of `@Controller`. It's essentially doing the same thing, except the `@ResponseBody` annotation is used automatically to annotate method results as the response body.

The `PostRestController` has a `PostRepository` field which gets populated in the class' constructor where the `@Autowired` annotation tells Spring to automatically inject an instance of the `PostRepository` class as a constructor argument.

The following three methods are separate REST services that we can call from our browser (for example: http://localhost:8080/post/search?keyword=us). Note how the `@RequestMapping` annotation configures the context path used for the REST service.
By default, the result of these methods are serialized into JSON objects, as you can see that in below screenshot.

Rebuild the application (Build / Make Project menu item) to refresh the running application with the latest code changes.


http://local...h?keyword=us
localhost:8080/post/search?keyword=u
Search

```
[{"id":1,"title":"Lorem ipsum dolor sit amet, consectetur adipiscing elit","body":"Fusce
urna nulla, fringilla lacinia euismod eget, vestibulum id metus."},{"id":2,"title":"Mauris
vulputate massa ac volutpat fermentum","body":"Ut at mollis purus, vitae feugiat lacus."},
{"id":3,"title":"Morbi porttitor pharetra ex nec eleifend","body":"Cum sociis natoque
penatibus et magnis dis parturient montes, nascetur ridiculus mus."},
{"id":4,"title":"Praesent iaculis sollicitudin ligula et lacinia","body":"Morbi commodo,
erat sit amet lobortis molestie, nibh mi tristique ligula, consectetur varius nisi erat in
orci."}]
```

## Creating Rest Resources

In the previous section we looked at how to create REST services using the `@RestController` annotation. Now we take a peek at the Spring Data Rest framework that allows to expose Spring Data repository interfaces (ie: `PostRepository`) as REST services.

First, add the spring-boot-starter-data-rest dependency to the `pom.xml` file under build/dependencies (around line 50):

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

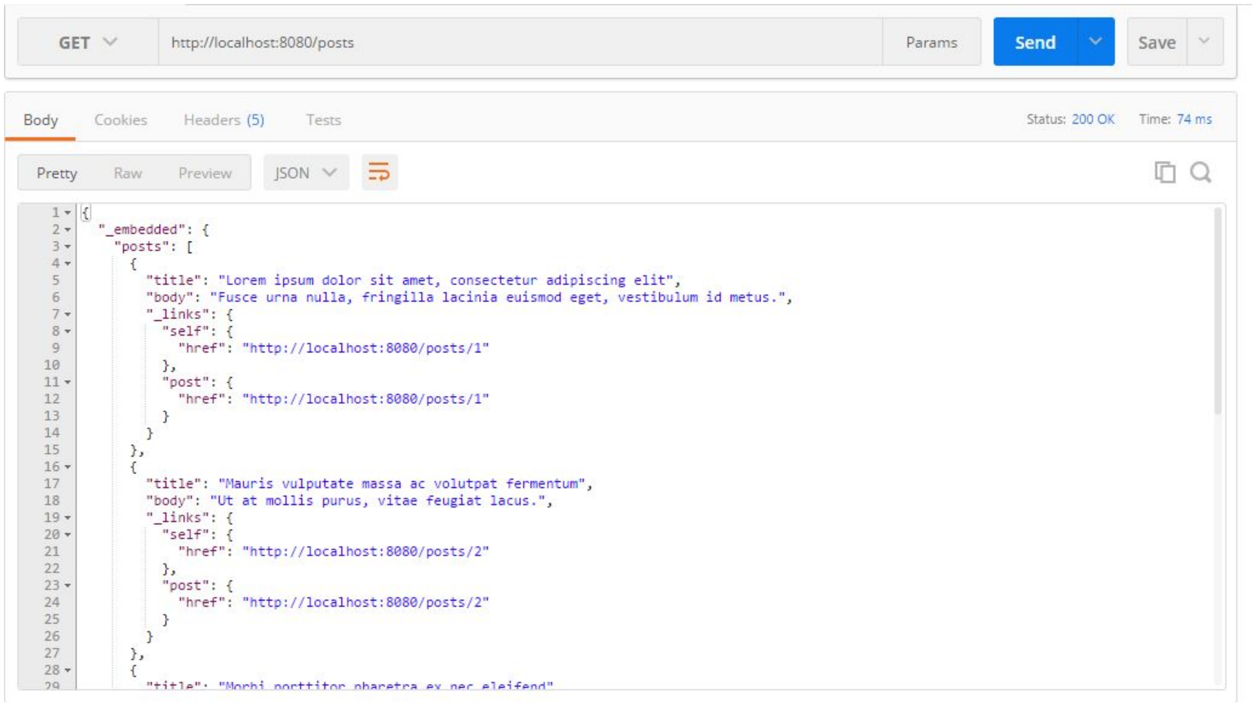
Step #2 is to replace the `@Repository` annotation with `@RepositoryRestResource` in the existing `PostRepository` class and add the `@Param` annotation to the method parameters which will set the name of the request parameters of the new REST services:

```java
package com.kaviddiss.repository;
```

```java
import com.kaviddiss.domain.Post;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import java.util.List;

@RepositoryRestResource(collectionResourceRel = "posts", path = "posts")
public interface PostRepository extends CrudRepository<Post, Long> {
   Post findOneByTitle(@Param("title") String title);

   List<Post> findByBodyContaining(@Param("keyword") String keyword);
}
```

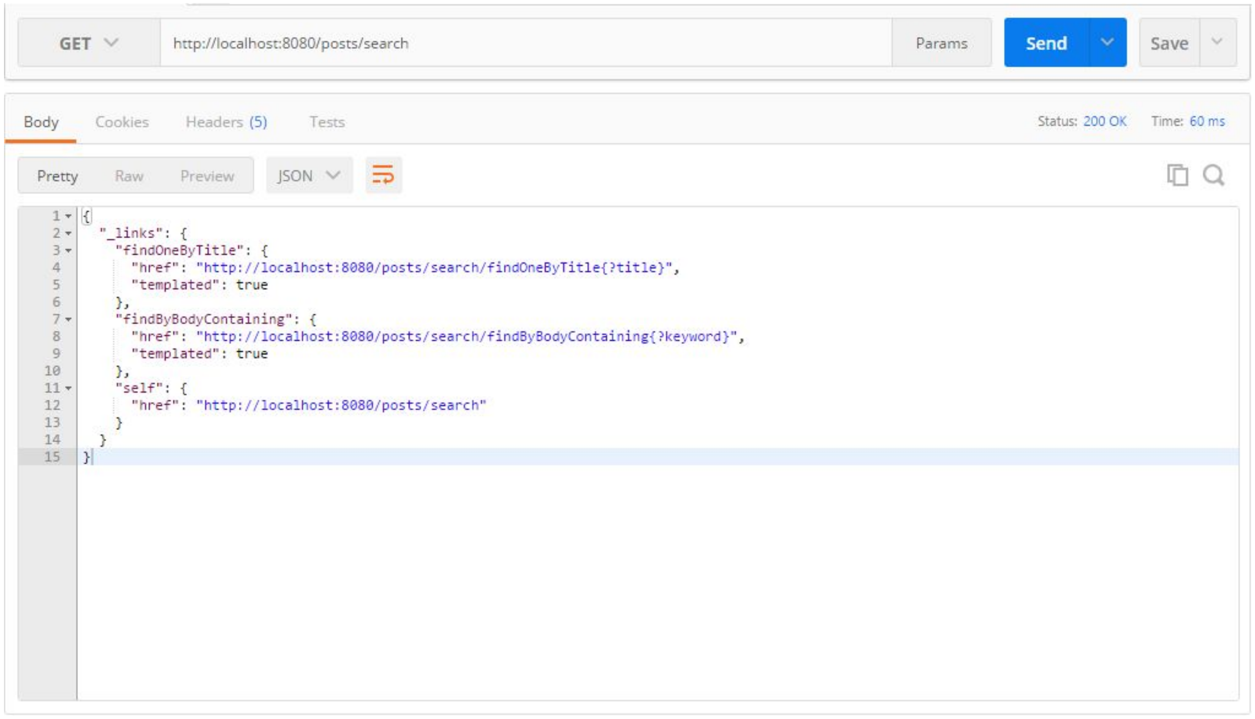
Rebuild or restart the running project for the changes to take effect.

If you have Chrome installed on your machine, install the Postman extension (offered by www.getpostman.com) to test the new REST services:



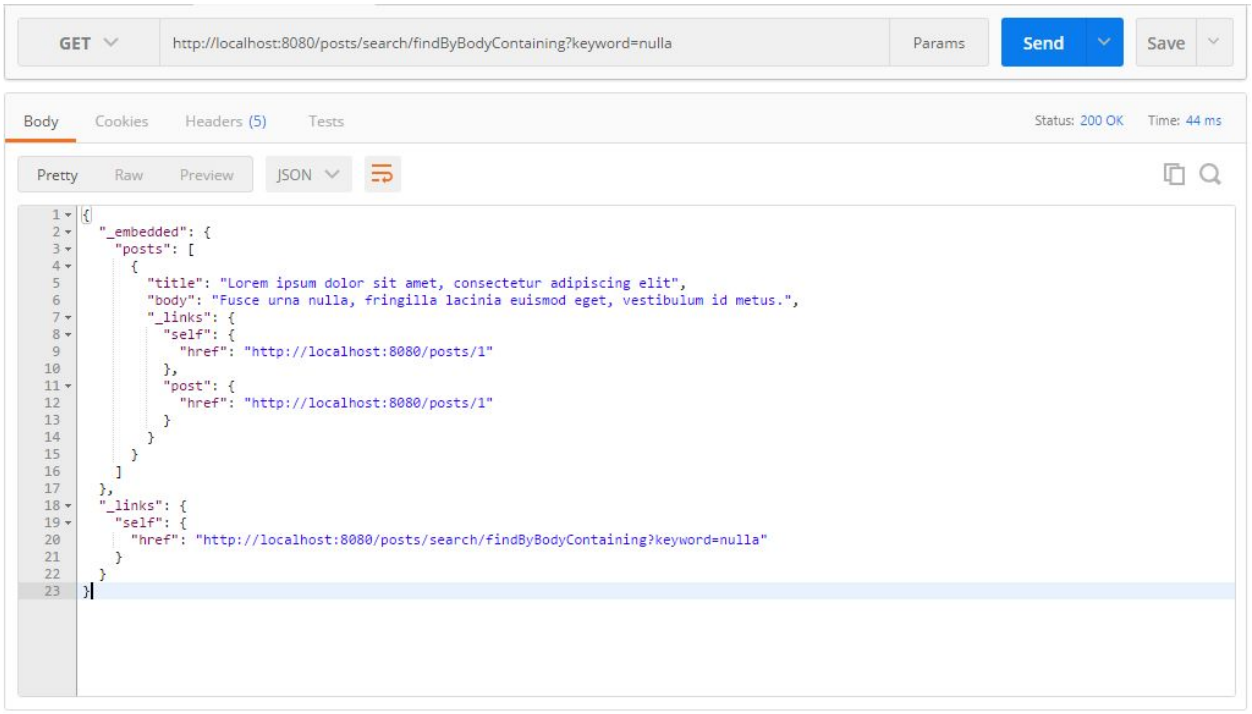
```
{
  "_embedded": {
    "posts": [
      {
        "title": "Lorem ipsum dolor sit amet, consectetur adipiscing elit",
        "body": "Fusce urna nulla, fringilla lacinia euismod eget, vestibulum id metus.",
        "_links": {
          "self": {
            "href": "http://localhost:8080/posts/1"
          },
          "post": {
            "href": "http://localhost:8080/posts/1"
          }
        }
      },
      {
        "title": "Mauris vulputate massa ac volutpat fermentum",
        "body": "Ut at mollis purus, vitae feugiat lacus.",
        "_links": {
          "self": {
            "href": "http://localhost:8080/posts/2"
          },
          "post": {
            "href": "http://localhost:8080/posts/2"
          }
        }
      },
      {
        "title": "Morbi porttitor pharetra ex nec eleifend"
```

*List of posts:* `curl -X GET "http://localhost:8080/posts"`



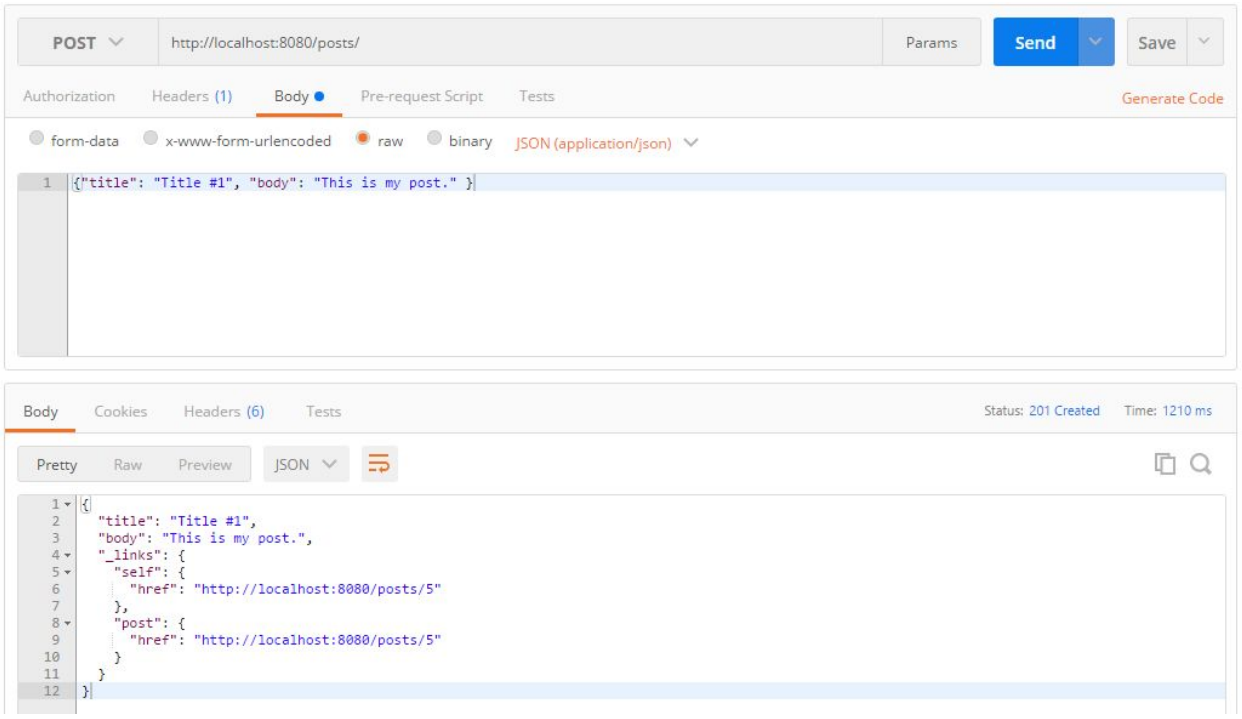
```
{
  "_links": {
    "findOneByTitle": {
      "href": "http://localhost:8080/posts/search/findOneByTitle{?title}",
      "templated": true
    },
    "findByBodyContaining": {
      "href": "http://localhost:8080/posts/search/findByBodyContaining{?keyword}",
      "templated": true
    },
    "self": {
      "href": "http://localhost:8080/posts/search"
    }
  }
}
```

*List of post search services:* `curl -X GET "http://localhost:8080/posts/search"`



```
{
  "_embedded": {
    "posts": [
      {
        "title": "Lorem ipsum dolor sit amet, consectetur adipiscing elit",
        "body": "Fusce urna nulla, fringilla lacinia euismod eget, vestibulum id metus.",
        "_links": {
          "self": {
            "href": "http://localhost:8080/posts/1"
          },
          "post": {
            "href": "http://localhost:8080/posts/1"
          }
        }
      }
    ]
  },
  "_links": {
    "self": {
      "href": "http://localhost:8080/posts/search/findByBodyContaining?keyword=nulla"
    }
  }
}
```

*Searching posts by keyword in body:* `curl -X GET "http://localhost:8080/posts/search/findByBodyContaining?keyword=us"`



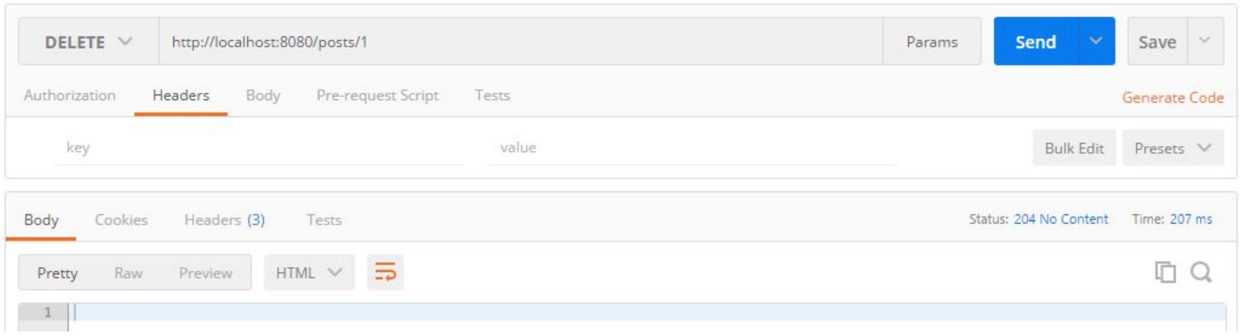
http://localhost:8080/posts/

```
{"title": "Title #1", "body": "This is my post." }
```

Status: 201 Created Time: 1210 ms

```
{
  "title": "Title #1",
  "body": "This is my post.",
  "_links": {
    "self": {
      "href": "http://localhost:8080/posts/5"
    },
    "post": {
      "href": "http://localhost:8080/posts/5"
    }
  }
}
```

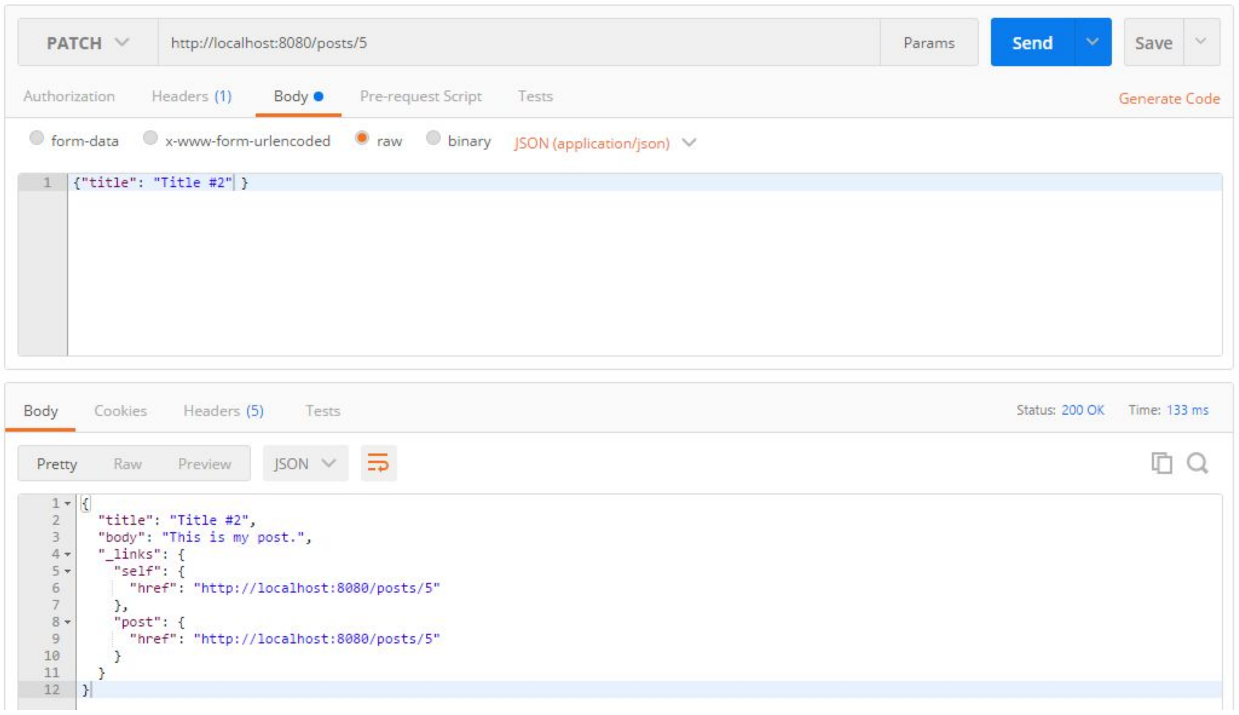
*Creating a new post:* `curl -X POST -H "Content-Type: application/json" -d '{"title": "Title #1", "body": "This is my post." }' "http://localhost:8080/posts/"`



http://localhost:8080/posts/1

Status: 204 No Content Time: 207 ms

*Deleting a post:* `curl -X DELETE "http://localhost:8080/posts/1"`



```
http://localhost:8080/posts/5
```

```
{"title": "Title #2" }
```

```
{
  "title": "Title #2",
  "body": "This is my post.",
  "_links": {
    "self": {
      "href": "http://localhost:8080/posts/5"
    },
    "post": {
      "href": "http://localhost:8080/posts/5"
    }
  }
}
```

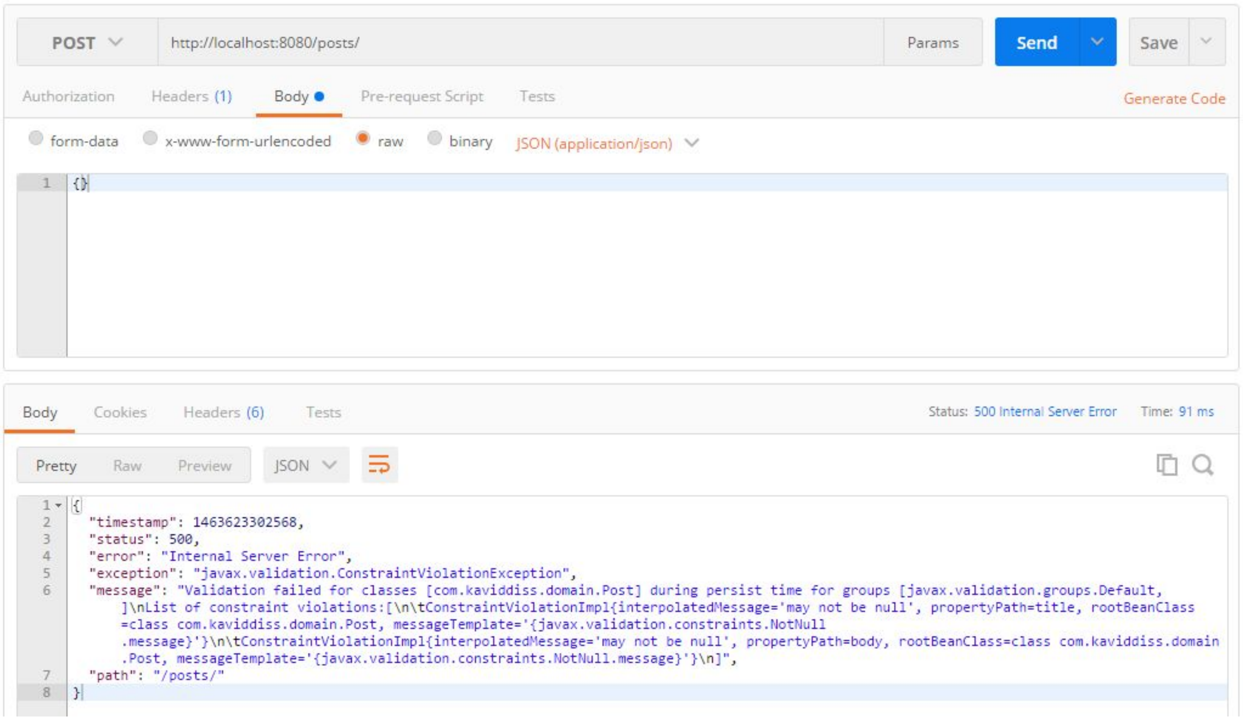
*Updating an existing post:* `curl -X PATCH -d '{"title": "Title #2" }' "http://localhost:8080/posts/5"`

Let's add some validation to make it a bit more realistic. Update the Post entity with the `@NotNull` annotation on both title and body fields:

```
package com.kaviddiss.domain;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.validation.constraints.NotNull;
import java.io.Serializable;

@Entity
public class Post implements Serializable {
   @Id
   @GeneratedValue
   private Long id;
   @NotNull
   private String title;
   @NotNull
   private String body;

   public Long getId() {
       return id;
   }

   public void setId(Long id) {
```

```java
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getBody() {
        return body;
    }

    public void setBody(String body) {
        this.body = body;
    }
}
```

Now let's see what happens if we don't provide any data (after rebuilding/restarting the application):



http://localhost:8080/posts/

```
{}
```

Status: 500 Internal Server Error Time: 91 ms

```
{
  "timestamp": 1463623302568,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "javax.validation.ConstraintViolationException",
  "message": "Validation failed for classes [com.kaviddiss.domain.Post] during persist time for groups [javax.validation.groups.Default,
      ]\nList of constraint violations:[\n\tConstraintViolationImpl{interpolatedMessage='may not be null', propertyPath=title, rootBeanClass
      =class com.kaviddiss.domain.Post, messageTemplate='{javax.validation.constraints.NotNull
      .message}'}\n\tConstraintViolationImpl{interpolatedMessage='may not be null', propertyPath=body, rootBeanClass=class com.kaviddiss.domain
      .Post, messageTemplate='{javax.validation.constraints.NotNull.message}'}\n]",
  "path": "/posts/"
}
```

*Creating a post without any data:* `curl -X POST -d '{}' "http://localhost:8080/posts/"`

When we tried to persist data using Spring Data, Spring automatically picks up the `@NotNull` annotations and validates the entity object against them

## Step #6 - Securing the application

In the previous section we covered how to create REST services that access a SQL database, in this section we'll look into securing those REST services with the help of the Spring Security framework.

> **What is Spring Security?**
> Spring Security is a Spring framework for authentication and authorization

We'll configure Spring Security to restrict access to the application by requiring Basic Authentication on all HTTP requests and to only authenticate requests where username/password is `user/pwd123`.

> **What is Basic Authentication?**
> Basic Authentication is the simplest type of authentication where the username and password is provided in the header of every HTTP request

Update the pom.xml by including the `spring-boot-starter-security` dependency (around line 44):

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

Create the `com.kaviddiss.config` package and add a `SecurityConfig` class in it:

```
package com.kaviddiss.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authentication.builders.AuthenticationM
anagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurer
Adapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   @Override
protected void configure(HttpSecurity http) throws Exception {
   http.authorizeRequests() // allows restricting access
```

```
        .anyRequest().fullyAuthenticated() // any HTTP request has to be authenticated
        .and()
            .httpBasic() // authentication has to be done via HTTP Basic Auth mechanism
        .and()
            .csrf().disable() // disable CSRF which is enabled by default
    ;
}

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    auth
        .inMemoryAuthentication() // store valid credentials in memory
        .withUser("user").password("pwd123") // only valid username/password is
user/pwd123
            .roles("USER")  // and that user will have USER role
    ;
}


}
```

The `@Configuration` annotation tells Spring that this is a configuration class used in the Spring application and the `@EnableWebSecurity` annotation tells Spring to initialize Spring Security on startup time.

To be able to configure Spring Security, the `SecurityConfig` class needs to extend the `WebSecurityConfigurerAdapter` class and implement the `configure()` and `configureGlobal()` methods.

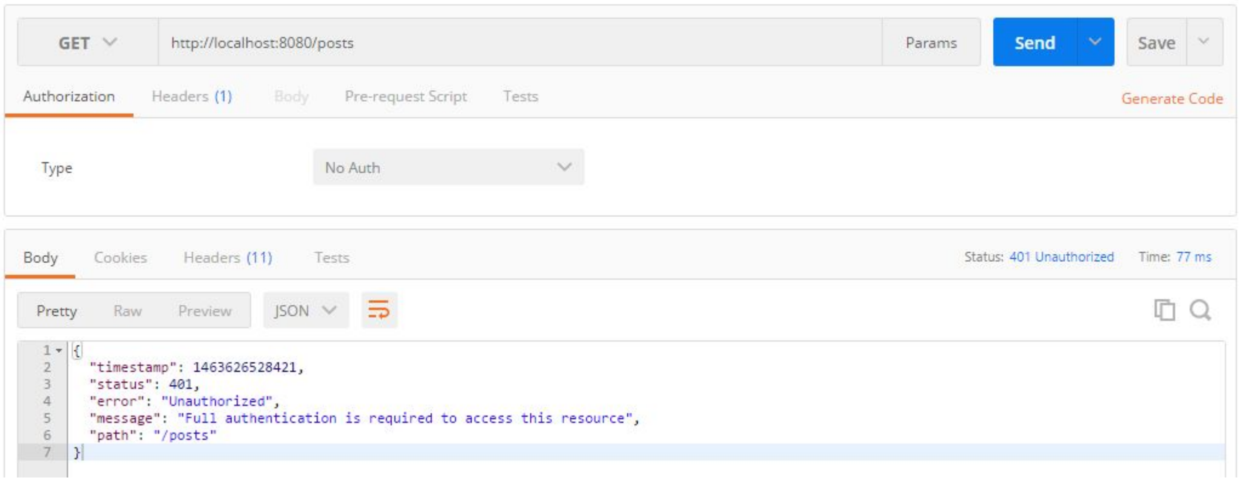
The `configure()` method tells Spring Security to require authentication on any HTTP request to the application using HTTP Basic Auth and disable protection for Cross-site request forgery (CSRF), though just for the sake of simplifying testing.

> **What is Cross-site request forgery?**
> “Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated”
> - https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

The `configureGlobal()` method tells Spring Security to only allow authentication for the hard-coded `user/pwd123` credentials stored in memory, and the logged in user will be granted with the USER role.

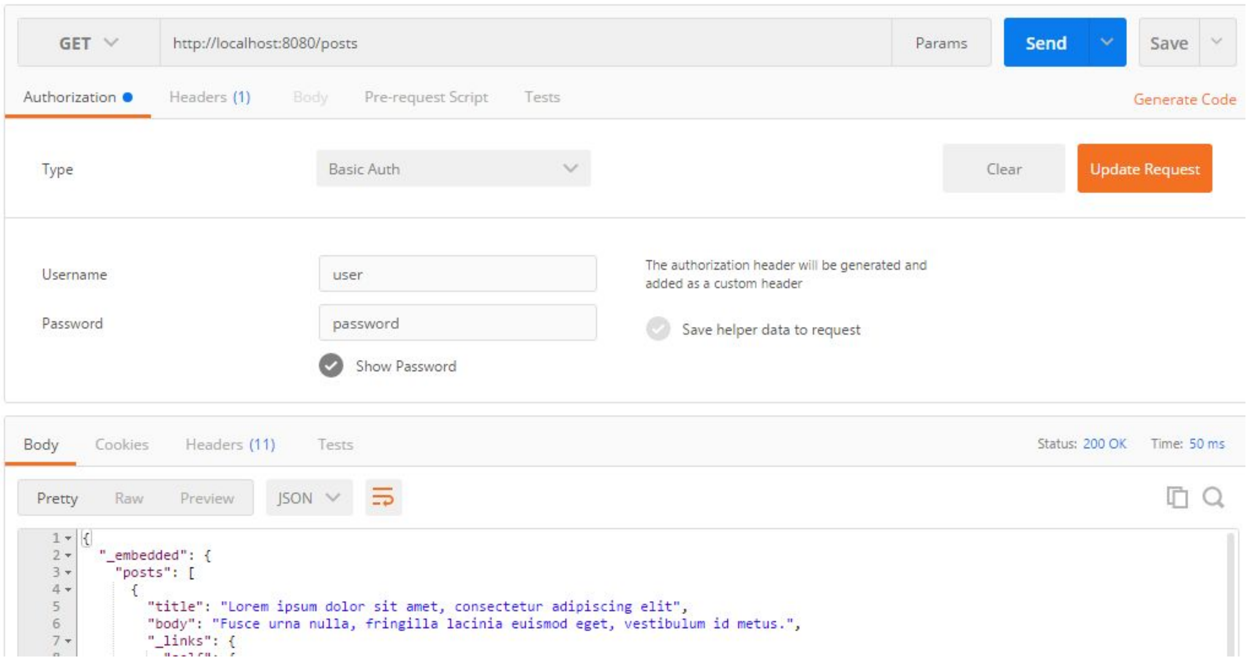
Note that in a production environment username, password and roles are most likely stored in an LDAP server or a database, but definitely not in-memory.

After rebuilding/restarting the application, let’s test these changes:



```
{
  "timestamp": 1463626528421,
  "status": 401,
  "error": "Unauthorized",
  "message": "Full authentication is required to access this resource",
  "path": "/posts"
}
```

*Unauthenticated request:* `curl -X GET "http://localhost:8080/posts"`



Type: Basic Auth

Username: user

Password: password

The authorization header will be generated and added as a custom header

```
{
  "_embedded": {
    "posts": [
      {
        "title": "Lorem ipsum dolor sit amet, consectetur adipiscing elit",
        "body": "Fusce urna nulla, fringilla lacinia euismod eget, vestibulum id metus.",
        "_links": {
```

*Authorized request:* `curl -X GET -H "Authorization: Basic dXNlcjpwYXNzd29yZA==" "http://localhost:8080/posts"`

## Conclusion

Congratulation on completing this tutorial! Now you mastered the basics of Spring.

If you’re wondering where to go next, you’ll find more tutorials on Spring at http://kaviddiss.com/learn-spring/.

In case you have any questions on this tutorial or on Spring in general, leave a message at http://kaviddiss.com/contact/. I’d also welcome any feedback on how to improve this tutorial.

Thanks!

David from http://kaviddiss.com