Spring Boot Hands-on Lab

Brian Clozel – bclozel@pivotal.io |

Table of Contents

During this Hands-on Lab, we'll create a simple URL shortener application, something widely used to share URLs on social media platforms (and track who's clicking on those links). You should have the previous Spring Boot Lab documentation handy, this could give you useful code samples.

> During the whole lab, you'll find the Spring Boot reference documentation (http://docs.spring.io/spring-boot/docs/2.0.6.RELEASE/reference/htmlsingle/) / javadoc API (http://docs.spring.io/spring-boot/docs/2.0.6.RELEASE/api/) and the Spring Framework reference documentation (http://docs.spring.io/spring-framework/docs/5.0.10.RELEASE/spring-framework-reference/) / javadoc API (http://docs.spring.io/spring-framework/docs/5.0.10.RELEASE/javadoc-api/) quite useful.

Bootstrap the Hands-on Lab application

Use the Spring Initializr (http://start.spring.io) to create your new application.

For this lab, we'll select the following options:

- We'll generate a **Maven** project with Spring Boot **2.0.6.RELEASE**
- Our group Id `fr.insalyon.tc` and artifact Id `spring-boot-hol`
- In the dependencies search box, select `Web` , `DevTools` and `Actuator` .

Click on "Generate", unzip the file somewhere, open it in your favorite IDE.

In the `spring-boot-hol` folder, your `pom.xml` file should look like this:

pom.xml

XML

```xml
<?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>fr.insalyon.tc</groupId>
    <artifactId>spring-boot-hol</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-boot-hol</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.6.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>  1
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>  2
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>  3
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>  4
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>


</project>
```

1 This starter brings the necessary dependencies for building a Spring MVC application

2 Actuator brings production ready features to your application (more on that later)

3 Devtools will improve your development experience

4 This starter brings useful dependencies for writing tests for your application

If you've got an IDE installed, you can import it as a Java application and run the `SpringBootHolApplication` class directly from the IDE.

Alternatively, you can open a new console and run the following command:

BASH

```bash
./mvnw spring-boot:run

[...]

2018-02-27 14:52:41.278  INFO 4247 --- [  restartedMain] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on
port(s): 8080 (http)
2018-02-27 14:52:41.283  INFO 4247 --- [  restartedMain] f.i.t.s.SpringBootHolApplication       : Started
SpringBootHolApplication in 2.665 seconds (JVM running for 2.968)
```

A new Tomcat server instance has started and our application is running on port `8080`. Using curl or a browser, we can send an HTTP request to `http://localhost:8080`.

BASH

```bash
$ curl http://localhost:8080/  1
{"timestamp":1519739778651,"status":404,"error":"Not Found","message":"No message available","path":"/"}%  2
```

1 a server is running on port 8080 (port chosen by default, but you can configure that)

2 since we've defined no web endpoint, Spring answers with a custom HTTP 404 error message

During the whole Hands-on lab, you can extensively use the previous Spring Boot Lab for code samples

Persisting short URLs to a MongoDB database

Spring Data helps you to manage Java objects and persist them in various data stores, such as Redis, PostgreSQL, ElasticSearch, MongoDB and many others - all using the same, consistent programming model while still using the specifics of each database.

In this lab, we'll use MongoDB; add the following dependencies to your application:

pom.xml

XML

```xml
<!-- Insert this snippet in the dependencies section -->
<dependency>
  <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>  1
</dependency>
<dependency>
  <groupId>de.flapdoodle.embed</groupId>
  <artifactId>de.flapdoodle.embed.mongo</artifactId>  2
</dependency>
```

1 Spring Data for MongoDB

2 For this lab, we'll use this in memory, embedded MongoDB database

First, we need to create an entity we can use and persist in MongoDB:

src/main/java/fr/insalyon/tc/springboothol/ShortURL.java

JAVA

```java
package fr.insalyon.tc.springboothol;

import java.net.URI;
import java.time.LocalDateTime;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.index.Indexed;
import org.springframework.data.mongodb.core.mapping.Document;

@Document  1
public class ShortURL {  2

    @Id  3
    private String id;

    private URI uri;  4

    @Indexed(unique = true)
    private String shortCode;  5

    private LocalDateTime creationDate;  6

    public ShortURL() {
    }

    public ShortURL(URI uri, String shortCode) {
        this.uri = uri;
        this.shortCode = shortCode;
        this.creationDate = LocalDateTime.now();
    }

    public String getId() {
        return id;
    }

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

    public URI getUri() {
        return uri;
    }

    public void setUri(URI uri) {
        this.uri = uri;
    }

    public String getShortCode() {
        return shortCode;
    }

    public void setShortCode(String shortCode) {
        this.shortCode = shortCode;
    }

    public LocalDateTime getCreationDate() {
        return creationDate;
    }

    public void setCreationDate(LocalDateTime creationDate) {
        this.creationDate = creationDate;
    }

    @Override
    public String toString() {
        return "ShortURL{" +
                "id='" + id + '\'' +
                ", uri=" + uri +
                ", shortCode='" + shortCode + '\'' +
                ", creationDate=" + creationDate +
                '}';
    }
}
```

1 To persist a class in a MongoDB repository, we need to declare it as a `@Document`
2 We call `ShortURL` the entity that holds the information about our shortened URL
3 We use a technical id, generated by the MondoDB database when inserted into the collection
4 The URI that our users want to shorten
5 The "short code" is the short representation of our URL; we annotate that attribute because we want it to be unique in our collection
6 We also store when the shortened URL was first created

Now to store/retrieve data from the database, Spring Data is using repositories. A repository is resposible for all those actions for a single document type. So for `ShortURL`, we'll need a `ShortURLRepository`.

With Spring Data, you don't need to implement all those classes; creating an interface that adheres to set a rules is enough for Spring Data to do all that for you.

src/main/java/fr/insalyon/tc/springboothol/ShortURLRepository.java

```java
package fr.insalyon.tc.springboothol;

import java.util.Optional; 1

import org.springframework.data.repository.CrudRepository; 2

public interface ShortURLRepository extends CrudRepository<ShortURL, String> { 3

  // add custom methods here 4
}
```

1 Instead of dealing with `null`, repositories are using `Optional` (check out its javadoc (https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html))
2 Our repository inherits from `CrudRepository`, take a look at the available methods there (https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#repositories.core-concepts). Don't worry too much about parameterized types `T, S, ID, etc`; just think of `T` and `S` as `ShortURL` and `ID` as `String`.
3 This repository will deal with `ShortURL` domain classes and their ID is of type `String`
4 We already inherit from `CrudRepository` and get many things for free, but we can also add our own methods here (more on that later)

Now, let's try adding a few ShortURLs to our database and fetching them back. For that, we'll use an `ApplicationRunner` that Spring Boot will run when our application starts:

src/main/java/fr/insalyon/tc/springboothol/RepositorySample.java

JAVA

```java
package fr.insalyon.tc.springboothol;

import java.net.URI;
import java.util.Arrays;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;

@Component
public class RepositorySample implements ApplicationRunner {

  private final Logger logger = LoggerFactory.getLogger(RepositorySample.class);


  @Override
  public void run(ApplicationArguments args) throws Exception {
    // TODO:
    // 1) create a few ShortURL objects
    // 2) persist them using the repository
    // 3) find all ShortURLs in the database
    // 4) log them using the logger
    }

}
```

Depending on how you implement that, you could see something like this in your console:

BASH

```bash
f.i.t.springboothol.RepositorySample   : short URLS :[ShortURL{id='5a987db3adf0cc26e92778ca',
uri=https://spring.io/blog, shortCode='spring', creationDate=2018-03-01T23:24:51.837},
ShortURL{id='5a987db3adf0cc26e92778cb', uri=https://www.qwant.com/?q=spring%20boot&t=all, shortCode='search',
creationDate=2018-03-01T23:24:51.837}, ShortURL{id='5a987db3adf0cc26e92778cc', uri=https://moodle.insa-lyon.fr,
shortCode='moo', creationDate=2018-03-01T23:24:51.837}]
```

Creating a dedicated ShortURLService

Nice, we can now save/fetch data from our database. But we still have a few issues, we want to:

- create random short codes for our users that are made of letters and numbers
- find a `ShortURL` instance by its short code, our repository doesn't allow that yet
- have all that in a tested Spring Component

First, you can copy/paste this class in your project:

src/main/java/fr/insalyon/tc/springboothol/ShortURLService.java

```java
package fr.insalyon.tc.springboothol;

import java.net.URI; // 1
import java.util.Random;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.util.Assert;

public class ShortURLService {

    private static final String SPACE = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-_"; // 2

    private static final int BASE = SPACE.length();

    /**
     * Create a new {@link ShortURL} using the given URI and save it
     * in a datastore. The {@code shortCode} used to identify this
     * URI should be randomly selected.
     * @param uri the URI to shorten
     * @return the shortened URL information
     */
    public ShortURL shortenURL(URI uri) {
        // TODO: create a ShortURL using a random positive integer
        // use java.util.Random and java.lang.Math for that
    // then use that random number with the provided encode method
        // Once created, persist that ShortURL into our database
    }

    /**
     * Return the {@link ShortURL} information for the given
     * {code shortCode}, if it exists in the datastore.
     * @param shortCode the shortened URL code to look for
     * @return the ShortURL information for the given shortCode
     * @throws ShortCodeNotFoundException if no ShortURL could be found
     * for the given shortCode
     */
    public ShortURL expandShortCode(String shortCode) {
        // TODO: find a ShortURL by its shortcode in the database and return it // 3
        // if there's none, throw a ShortCodeNotFoundException
    }

    private static String encode(int num) { // 4
        Assert.isTrue(num > 0, "Number must be positive");
        StringBuilder str = new StringBuilder();
        while (num > 0) {
            str.insert(0, SPACE.charAt(num % BASE));
            num = num / BASE;
        }
        return str.toString();
    }

}
```

1 Don’t know which part of the Java library you should use? Look at the provided imports here…

2 we’ll create short codes using the following characters

3 You’ll need to add a new method to your repository to find a ShortURL by its short code; check out the Spring Data reference documentation for that (https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#repositories.query-methods.query-creation)

4 given a positive number, this will return a short code

Here’s the exception we mentioned above:

src/main/java/fr/insalyon/tc/springboothol/ShortCodeNotFoundException.java

```java
package fr.insalyon.tc.springboothol;

public class ShortCodeNotFoundException extends RuntimeException {

    private final String shortCode;

    public ShortCodeNotFoundException(String shortCode) {
        super(String.format("Could not find ShortURL for [%s]", shortCode));
        this.shortCode = shortCode;
    }

    public String getShortCode() {
        return shortCode;
    }
}
```

To help you implement the previous methods, you can use the following test class. Run the tests with `./mvnw test` and fix your code until all tests pass.

src/test/java/fr/insalyon/tc/springboothol/ShortURLServiceTests.java

```java
package fr.insalyon.tc.springboothol;


import java.net.URI;
import java.util.Optional;

import org.hamcrest.Matchers;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;
import org.mockito.Mock;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.ArgumentMatchers.eq;
import static org.mockito.BDDMockito.given;
import static org.mockito.Mockito.doAnswer;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.MockitoAnnotations.initMocks;

public class ShortURLServiceTests {

    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Mock
    private ShortURLRepository repository;

    private ShortURLService shortURLService;

    @Before
    public void setup() {
        initMocks(this);
        this.shortURLService = new ShortURLService(this.repository);
        doAnswer(invocation -> invocation.getArgument(0)).when(this.repository).save(any());
    }

    @Test
    public void canShortenURLs() {
        URI uri = URI.create("http://example.org/resource");
        ShortURL shortURL = this.shortURLService.shortenURL(uri);
        assertThat(shortURL.getShortCode()).isNotBlank();
        assertThat(shortURL.getUri()).isEqualTo(uri);
        verify(this.repository, times(1)).save(any());
    }

    @Test
    public void canExpandShortCodes() {
        URI uri = URI.create("http://example.org/resource");
        ShortURL shortURL = new ShortURL(uri, "spring");
        given(this.repository.findByShortCode("spring")).willReturn(Optional.of(shortURL));
        ShortURL result = this.shortURLService.expandShortCode("spring");
        assertThat(result.getUri()).isEqualTo(uri);
        assertThat(result.getShortCode()).isEqualTo("spring");
        verify(this.repository, times(1)).findByShortCode(eq("spring"));
    }

    @Test
    public void unknownShortCode() {
        given(this.repository.findByShortCode("spring")).willReturn(Optional.empty());
        this.thrown.expect(ShortCodeNotFoundException.class);
        this.thrown.expect(Matchers.hasProperty("shortCode", Matchers.equalTo("spring")));
        ShortURL result = this.shortURLService.expandShortCode("spring");
    }

}
```

Exposing that service on the Web

Now that we have a `ShortURLService` component that we can reuse, we'd like to expose that as a web service so that:

- users can shorten URLs by sending links to the service
- the application will redirect short URLs to the real ones

Spring MVC is a Web Framework that can help you build web applications; you'll be mostly dealing with Controllers while building your application.

What are Spring MVC Controllers:

- they're regular Spring components, annotated with `@Controller`
- they are the adapting layer between the HTTP world and your other Spring components

src/main/java/fr/insalyon/tc/springboothol/ShortURLController.java

JAVA

```java
package fr.insalyon.tc.springboothol;


import java.net.URI;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.servlet.mvc.method.annotation.MvcUriComponentsBuilder;

import static org.springframework.web.servlet.mvc.method.annotation.MvcUriComponentsBuilder.on;

@Controller  1
public class ShortURLController {

private final ShortURLService shortURLService;

    @Autowired
    public ShortURLController(ShortURLService shortURLService) {  2
    this.shortURLService = shortURLService;
    }

    @PostMapping("/")  3
    public ResponseEntity shortenURL(@RequestParam URI uri) {  4
        ShortURL shortURL = this.shortURLService.shortenURL(uri);
        URI shortenedURL = URI.create("http://localhost:8080/" + shortURL.getShortCode());
        // the host information is hard coded here for simplicity
        // but you can do better with MvcUriComponentsBuilder::fromMethodCall
        return ResponseEntity.created(shortenedURL).build();  5
    }

}
```

1. This class is declared as a Controller, which is a Spring component.
2. As any other, you can inject it with other components that you need.
3. The `@PostMapping`, `@GetMapping`, `@PutMapping` etc. annotations tell Spring MVC what kind of HTTP requests should be handled by your controller methods. Here, we're mapping POST requests to "/".

4 Method parameters can be annotated to tell Spring MVC to extract information from the HTTP request and inject them here for you. Here, we're asking to inject here an HTTP request param called "uri".

5 Spring MVC will take the returned value and create an HTTP response with it. Depending on the return type, the strategy will be different. Here, with a `ResponseEntity`, we're taking full control over the HTTP response status, headers, content.

Run this application and try to send an HTTP request to that endpoint; we've just saved a ShortURL in our database!

BASH

```
$ curl -d uri=http://example.org http://localhost:8080/ -v

> POST / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
> Content-Length: 22
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 22 out of 22 bytes
< HTTP/1.1 201
< Location: http://localhost:8080/1giL2D
< Content-Length: 0
<
```

Redirecting existing ShortURLs

Now it's your turn to add a new endpoint to our Controller. When a user requests a shortened URL, we want to permanently redirect them to the real URL.

This should behave like this:

BASH

```
curl localhost:8080/1giL2D -v

> GET /1giL2D HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 308
< Location: http://example.org
< Content-Length: 0
<
```

For that, you'll need to take a look at the Spring MVC handler methods reference documentation (https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-ann-methods) to see how you should write your method signature.

In this particular case, you'll need `@PathVariable` and `@GetMapping` to get the job done. You'll also need to return a `ResponseEntity` again, but this time using other methods to set the location header and the response status (see javadoc (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/ResponseEntity.html)).

Getting info on ShortURL in JSON format

Now we'd like to provide information about existing `ShortURL` in JSON format. We want to create an endpoint that should behave like this:

BASH

```
curl localhost:8080/1giL2D/info
{"id":"5a988057adf0cc277a11bd92","uri":"http://example.org","shortCode":"1giL2D","creationDate":"2018-03-
01T23:36:07.356"}%
```

In this case, we'd like Spring MVC to transform our `ShortURL` object into JSON. Check out again the handler methods documentation and especially `@ResponseBody`.

Testing your MVC endpoints

Now that everything works, we'd like to check that all the features are implemented properly.

Use this class to verify that your Controller methods are right:

src/test/java/fr/insalyon/tc/springboothol/ShortURLControllerTests.java

```java
package fr.insalyon.tc.springboothol;

import java.net.URI;

import org.hamcrest.Matchers;
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import static org.mockito.ArgumentMatchers.eq;
import static org.mockito.BDDMockito.given;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.header;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@RunWith(SpringRunner.class)
@WebMvcTest(ShortURLController.class)
public class ShortURLControllerTests {

  @Autowired
  private MockMvc mvc;

  @MockBean
  private ShortURLService shortURLService;

  @Test
  public void shouldShortenURL() throws Exception {
    URI uri = URI.create("http://example.org");
    ShortURL shortUrl = new ShortURL(uri, "example");
    given(this.shortURLService.shortenURL(eq(uri))).willReturn(shortUrl);
    this.mvc.perform(post("/").param("uri", "http://example.org"))
        .andExpect(status().isCreated())
        .andExpect(header().string("Location", Matchers.endsWith("/example")));
    }

  @Test
  public void shouldRedirectToURL() throws Exception {
    URI uri = URI.create("http://example.org");
    ShortURL shortUrl = new ShortURL(uri, "example");
    given(this.shortURLService.expandShortCode(eq("example")))
        .willReturn(shortUrl);
    this.mvc.perform(get("/example"))
        .andExpect(status().isPermanentRedirect())
        .andExpect(header().string("Location", "http://example.org"));
  }

  @Test
  public void shouldReturnNotFoundForMissingShortURLs() throws Exception {
    given(this.shortURLService.expandShortCode(eq("example")))
        .willThrow(new ShortCodeNotFoundException("example"));
    this.mvc.perform(get("/example"))
         .andExpect(status().isNotFound());
  }

  @Test
  public void shouldShowJSONInfo() throws Exception {
    URI uri = URI.create("http://example.org");
    ShortURL shortUrl = new ShortURL(uri, "example");
    given(this.shortURLService.expandShortCode("example")).willReturn(shortUrl);
    this.mvc.perform(get("/example/info").accept(MediaType.APPLICATION_JSON))
```

```
        .andExpect(status().isOk())
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.shortCode").value("example"))
        .andExpect(jsonPath("$.uri").value("http://example.org"));
  }

}
```

If the `shouldReturnNotFoundForMissingShortURLs` test fails for you, see how @ResponseStatus (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseStatus.html) can be added to your exception class.

Listing all ShortURLs in an HTML view

Spring MVC also helps you creating dynamic HTML pages for your application:

- Spring MVC can render templates (think: dynamic HTML files) with data you put in the Model map
- instead of returning directly the response body, your Controller methods can return the name of the view as a `String`

We'll use the Mustache templating engine for this, so add the following dependency to your build:

pom.xml

XML

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

Also, add the following Mustache template to your application and see how it iterates over all ShortURLs stored in a variable called "urls":

src/main/resources/templates/list.mustache

HTML

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>All ShortURLs</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.6.2/css/bulma.min.css">
    <script defer src="https://use.fontawesome.com/releases/v5.0.6/js/all.js"></script>
</head>
<body>
<section class="section">
    <div class="container">
        <h1 class="title">
            List of all ShortURLs
        </h1>
        <table class="table is-striped is-hoverable is-fullwidth">
            <thead>
            <tr>
                <th>ShortCode</th>
                <th>URL</th>
                <th>Creation Date</th>
            </tr>
            </thead>
            <tbody>
            {{#urls}}
            <tr>
                <td>{{shortCode}}</td>
                <td><a href="{{uri}}">{{uri}}</a></td>
                <td>{{creationDate}}</td>
            </tr>
            {{/urls}}
            </tbody>
        </table>
    </div>
</section>
</body>
</html>
```

Now do the following:

1. Add a new method to your `ShortURLService` that returns all instances stored in our database
2. Create a new Controller method that uses the `"list"` view and add the list of all URLs to the model map (you can map that view to GET requests on "/")

For more guidance, you can check this sample Spring Boot Mustache application (https://github.com/spring-projects/spring-boot/tree/master/spring-boot-samples/spring-boot-sample-web-mustache).

Have some spare time?

Try adding a new attribute on `ShortURL` that counts the number of times the short link has been accessed by people.

Manually incrementing the `ShortUrl` counter and saving it back into the database is not a great idea, since many people can access the same URL at the same time and you'll face concurrent modifications. Mongo offers specific `"$inc"` operations for that:

- Check out this part of Spring Data MongoDB reference documentation (https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#mongodb-template-update)
- Spring Boot creates automatically a `MongoTemplate` bean for you, that you can inject in your service layer

Last updated 2018-11-16 08:21:11 +0100