Offline Tutorial: Java Spring Framework – Concepts and Examples

Kevin Wang, kevinw@cse.ust.hk

August 2017

Why are we even reading this.

Throughout the course COMP3111 we are working on a chatbot (chatting-robot) on the instant messager software LINE. One of the easiest way to do is using the example provided by LINE developer site and it is written in Java Enterprise Edition (J2EE). In order to complete the project you will need to pick up a set of skills, which is already *minimized* when we design the project, as listed below:

1. git, *a version controlling repository tools.*
2. Java, *an object oriented programming language.*
3. JUnit, *a testing suite for Java.*
4. Database, *a program that manage your data.*
5. Spring, *a framework that provide web service in Java Enterprise Edition.*
6. Gradle, *a package management tools, like makefile, Ant, Maven.*

As you can see we said "minimized" but there are still a lot to cover. It is impossible to cover all these contents in the lecture or tutorial. Besides, you should also develop the skill to pick up new contents therefore we are here. This document is one of the seven topics tutorial and you are supposed to read this at home and you will be given chance to practise them at lab. In any case you have encounted difficulty in reading this document, you can post your question on the forum or contact the TA for help.

Contents

1 Background

Spring is a Java framework that mainly used for developing web applications on Java Enterprise Edition (J2EE) platform. You may wonder how would that be relevant to our project – a chatbot that replies LINE message. In fact when a user send a LINE message, it will be first caught by the LINE server and then relayed to your chatbot using HTTP protocol. The chatbot will also reply the LINE server using HTTP protocol. In order to perform this send-and-receive activities with the LINE server, a web application needs to be built.

Spring is a rather advanced and very comprehensive framework. We are unable to cover everything in this tutorial. You should have read the Basic Java tutorial prior to this document. Some references are provided in this document. They could be useful when you have any doubt.

In this tutorial all configurations are well handled by Spring boot and we are not doing any XML editing. However most of the web resources available get into the configuration very details. You don't need to worry about the configuration at the moment.

✍ **Learn More:**

- Code used in this tutorial Branch master is the starting code while branch complete is the complete code.
- Book – Beginning Spring
- Tutorial Points – Spring Tutorial

2 Concepts

2.1 HTTP

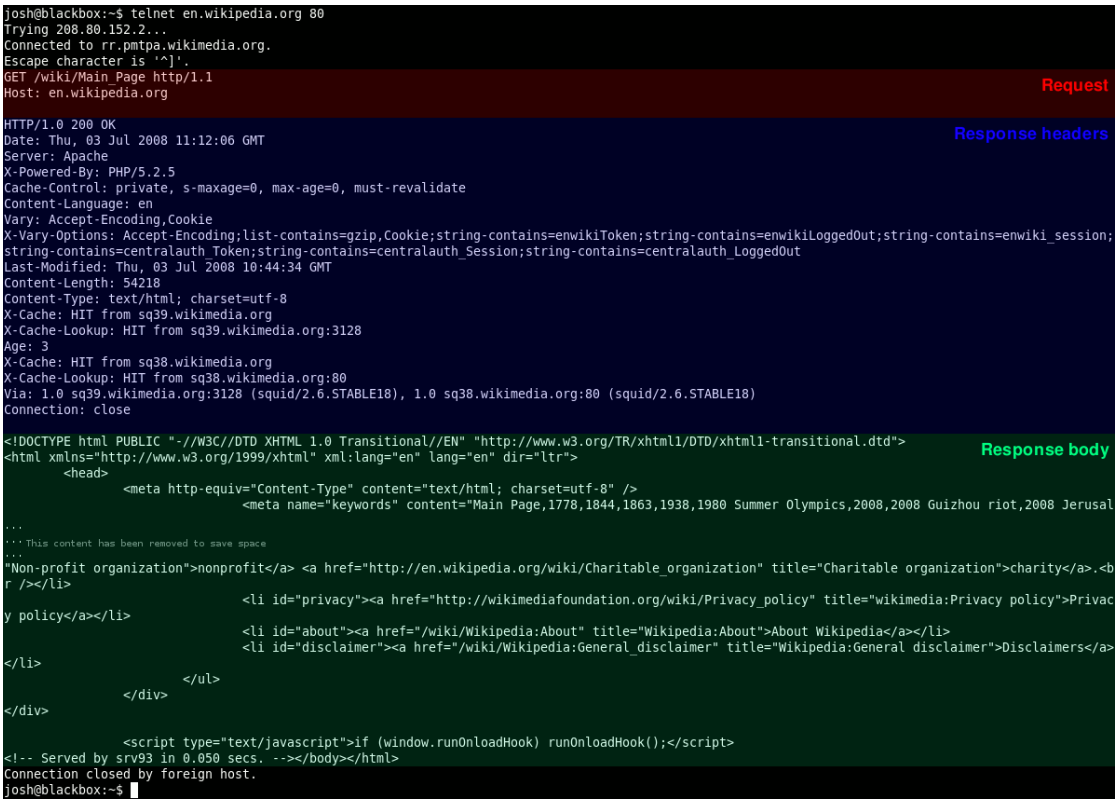
The network protocol HTTP defines a set of "languages" that allows a client (e.g. a web-browser) to communicate with a web-server. Typically after a client connected to a server, the client will issue a request by some *request methods* and the server will *response* with a *response header* and possibly some contents as its *response body*. Fig ?? shows an example of how HTTP works when a client try to access a web page.

Depends on different scenarios, the response header responsed from the server should contain different *status codes* which indicates the status of this request. For example, when a server response 404 that means a page is not found.

Common request methods include **GET, PUT, POST** etc. You shall note the differences between GET and POST method. In GET method parameters are send together with the URL like:

```
http://abc.com/requestPath?parameter1=x&parameter2=y
```

Figure 1: Sample of request method, response header and response body. Sourced from Wikipedia



which attempt to access the directory requestPath with two parameters. With POST method you need to embed the parameters in the header. The URL would looks like

```
http://abc.com/requestPath
```

while the header could be

```
POST /requestPath/ HTTP/1.1 Host:  abc.com parameter1=x&parameter2=y
```

The request methods are selected by the client. For example, you can use *curl* on linux/mac to request a webpage and you can select your request methods

```
# GET method
curl -X GET "http://abc.com/requestPath?parameter1=x\&parameter2=y"
# POST method
curl -d "parameter1=x\&parameter2=y" -X POST http://abc.com/requestPath
```

✍ **Learn More:**

- List of HTTP Request Methods
- List of HTTP Status Codes

2.2 JSON

JSON is a common data representation. It is very convenient for two machines to exchange data. The syntax of JSON looks like:

```
{
   "name": "John",
   "sex" : "Male"
 }
```

The LINE message API interface is using JSON in representing data and in this tutorial some messages are also represented in JSON.

2.3 Inversion of Control (IoC) / Dependency Injection

Consider the following scenario

```
class Foo {
    private Bar myBar;
    public Foo() {
        myBar = new Bar();
    }
}

class Bar {
// written by someone and is currently updating
}
```

We says *Foo* is depending on *Bar*. Someday *Foo* may become incompatible to the project because *Bar* removes the default constructor from its public interface. As a good software engineering practice, we want to remove this dependency.

```
class Foo {
    private Bar myBar;
    public Foo(Bar aBar) {
        myBar = aBar;
    }
}
```

Now, this is someone else's task to construct Bar and *inject* this dependency later to your class. Of course one may do this task in the main function. However, in Spring this is done via a proper configuration. For example, using a XML file like:

```
<beans>
    <bean id="foo" class="Foo">
        <constructor-arg ref="bar"/>
    </bean>

    <bean id="bar" class="Bar"/>
</beans>
```

So when the application starts Spring will construct the object and pass suitable argument to the constructor. We will look into this in next section.

✍ **Learn More:**

- What is Inversion of Control – Stackoverflow
- Spring IoC container

2.4 Gradle

In the most simply analogy to describe Gradle – a more powerful makefile commonly used in Java. The configuration file of gradle is written in the file "build.gradle". Similar to makefile, one may define tasks and their dependencies. What makes gradle so popular is that you can include a package with one line of configuration and gradle will automatically connect to a remote repository and download the correct/latest version of that package to build.

✍ **Learn More:**

- Tutorials Point - Gradle
- Building Spring Project with Gradle

3 Getting Started on Spring

Spring Boot is a tools that further simplifies configurations and allows developers to focus on developing their app. It has embedded a web server that responses to HTTP requests with your code running on it. You need about 15-30 minutes to complete the following guide.

3.1 Installation

You need to install Java SDK 1.8 64-bits, Eclipse STS 3.9.0 or above and then Gradle (STS) 3.8.x+1.0.x in order. They are installed at lab, avaliable on L:\apps\comp3111.

To install Gradle (STS) you need to start the Eclipse STS first. Click "Help" →"Eclipse Marketplace.." →and Type "Gradle IDE Pack" in the search box and install. You need to reboot your STS after install.

Next you need to checkout the project from the github by opening your console on linux/mac/windows. Make sure you have installed git if you are using Windows.

```
$ git clone https://github.com/khwang0/learning-spring.git
```

A project folder will be created at your machine.

3.2 Starting a Spring Boot project

Open the project in Eclipse STS. right click on the project and select "configure→Convert to Gradle", and press "Ctrl-Alt-Shift-R" to launch the "Task Quick Launcher". Type "bootRun" and press enter to launch the project. Then, open "localhost:8080/" in your browser and look at the result.

Now, let's look at the codes and appreciate how simple is it.

```
//Application.java
package hello;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

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

    }

}

//HelloController.java
package hello;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
```

```
public class HelloController {

    @RequestMapping("/")
    public String index() {
        return "hi";
    }

}
```

The configuration file build.gradle looks like

```
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'org.springframework.boot'

sourceCompatibility = 1.8
targetCompatibility = 1.8

buildscript {
 repositories {
    mavenCentral()
 }
 dependencies {
  classpath("org.springframework.boot:spring-boot-gradle-plugin:1.5.6.RELEASE")
 }
}
repositories {
 mavenCentral()
}

dependencies {
 compile("org.springframework.boot:spring-boot-starter-web")
 testCompile("org.springframework.boot:spring-boot-starter-test")
}
```

The core feature of Spring is Inversion-of-Control where classes are lossely coupled. In particular, *Application* and *HelloController* are not known to each other. Normally we need to specify which class is the controller that handle the request (e.g. using an XML file). However, spring boot saves our configuration work by scanning the directory and looking for the appropriate controller.

The annotation @RestController states HelloController is a RESTful controller and will accept and return parameters in JSON. The annotation @RequestMapping states the following function will handle requests on the root directory "/". It is very easy to map another request by stating another function and try the example below.

Suppose we want to send an object of class *Student* instead of just a String. We define the class somewhere in our project (a separated file or below *HelloController*).

```
class Student {
    private String name;
    private int year;
    public Student(String name, int year) {
        this.name = name;
        this.year = year;
    }
    public String getName() {
        return name;
    }
    public int getYear() {
        return year;
    }
}
```

Then we add another function into *HelloController*.

```
@RequestMapping("/hi")
 public Student sayHiToStudent() {
    return new Student("Kevin", 3);
}
```

When you run the project again and access the page "localhost:8080/hi", you will see:

```
 "name":"Kevin","year":3
```

DI allows you to link another component object without explicitly calling the constructor of that component. Suppose we add another Java file *SomeSpringBean.java* into the project:

```
package hello;

import org.springframework.stereotype.Component;

@Component
public class SomeSpringBean {
    private int counter = 0;
    public int getCounter() {return counter++;}
}
```

Inside your *HelloController* class, we change the code a little bit as

```
@Autowired
private SomeSpringBean myBean;
```

```
@RequestMapping("/")
public String index() {
     return "Greetings from Spring Boot!" + myBean.getCounter();
//  return "hi";
}
```

Once spring boot is started, the Autowired variable *myBean* will be wired by a class annotated with @Component. What spring does is to construct the SomeSpringBean object and *Inject* that to *HelloController*. You shall see the result below when you first run it. The number will increment per the number of refresh you press on the browser.

```
Greetings from Spring Boot!0
```

Now, we might want to add some parameters when we access this page. We change the handler function of "/hi" as

```
@RequestMapping("/hi")
public Student sayHiToStudent(
           @RequestParam(value="parameter1", defaultValue="Kevin")
            ↪  String name,
         @RequestParam(value="parameter2", defaultValue="") String who )
          ↪  {
               return new Student(name + " " + who, 3);
}
```

This function now takes two parameters from the request methods. The order is not really important as while *parameter1* will be mapped to *name* and *parameter2* will be mapped to *who*. Open the link http://localhost:8080/hi?parameter1=Spring&parameter2=Boot to test the result. You shall see:

```
"name":"Spring Boot","year":3
```

You can also restrict the request method. For instance, you need the client to send the password to your server and it is definitely a bad idea to embed the password on the URL. To can change the @RequestMapping annotation like

```
@RequestMapping(value="/hi", method=RequestMethod.POST)
public Student sayHiToStudent(
...
}
```

If you open the webpage again you shall see something like

```
Whitelabel Error Page

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

Tue Aug 08 11:10:56 CST 2017
There was an unexpected error (type=Method Not Allowed, status=405).
Request method 'GET' not supported
```

✍ **Learn More:**

- Getting Started Tutorial – Building a RESTful Web Service

3.3 Testing

Testing is a very important matter in software engineering. Spring has integrated some test suites for you to test your code. A very simple example illustrated below test our *HelloController* class. Create the file 'HelloControllerTester.java' under the path "/src/test/java/hello/" with the following content.

```
package hello;

import static org.assertj.core.api.Assertions.assertThat;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.context.embedded.LocalServerPort;
import org.springframework.boot.test.context.SpringBootTest;
import
 ↪  org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HelloControllerTester {

    @LocalServerPort
    private int port;

    @Autowired
```

```
    private TestRestTemplate restTemplate;

    @Test
    public void greetingShouldReturnDefaultMessage() throws Exception {
        assertThat(this.restTemplate.getForObject("http://localhost:" +
        ↪  port + "/", String.class))
                .contains("Greeting").doesNotContain("Error");


    }
}
```

To test your code you need to run the task "test" by "Ctrl-Alt-Shift-R" to launch the "Task Quick Launcher" and type "test". The program will stop when the test is completed. In case the test case does not pass, an error will be generated. For example, if we change the statement to

```
assertThat(this.restTemplate.getForObject("http://localhost:" + port +
↪  "/", String.class))
    .contains("GreetingABCDEF").doesNotContain("Error");
```

An error will be generated. Some red words will be displayed in the console:

```
1 test completed, 1 failed
:test FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':test'.
> There were failing tests. See the report at: /build/reports/tests/index.html
```

You can click on the file /build/reports/tests/index.html and see the detail diagnosis.

✍ **Learn More:**

- More syntax related to assertThat
- A better Tutorial to Spring test

✍ **Learn More:**

- The Spring Framework provide many nice examples for *Getting Starts*. You can click "File→New Project→Spring Boot→Import Spring Getting Start Content" to bootstrap your development.

4 Self Test

1. What is dependency injection?
2. What is a similar software to gradle that you have used in writing C++?
3. Based on the code above, try to write a test that assert the response is always starts with (not just contains) the word “Greeting” .
4. Based on the code above, handle the request on path /count that returns the sum of visits to the directory /, /hi, and /count.