

1. Tutorial Information

- Title: Building SpringFramework and Spring Boot Project
- Tutorial time: April 11 & 12, 2023
- Lecture Material: <http://www.kousenit.com/springboot/>
- Lecture Solution 1: <https://github.com/kousen/spring-and-spring-boot>
- Lecture Solution 2: <https://github.com/th-hsieh/LearningSpring-SpringBoot>
- Lecturer:
Ken Kousen
ken.kousen@kousenit.com
<http://kousenit.org>

2. Table of Contents

2-1. Creating a New Project

We build a Gradle project for a simple Spring MVC structure which has Model, Controller, and View java-src-wise. Testing-wise we have Unit test for Controllers, and then integration test using MockMVC test.

2-2. Add a Rest Controller

Add a HelloRestController.java and Greeting.java(a POJO). What we are doing here is simply sending an HTTP request(by ourselves manually, we are hitting the URL by ourselves and pretend we're clients haha) to the URL "/rest" And the RestController handles this request with @RequestParam and returns a response with a new Greeting object(which is defined in another class, Greeting.java).

2-3. Building a REST client

This exercise uses the new **reactive web client(WebClient)** to access a RESTful web service. The template converts the response into an object for the rest of the system. Older Spring application, RestTemplate, is being gradually replaced with WebClient. **WebClient returns responses of type Mono and Flux.** They are "promises" that return a single object (for Mono) or a collection (for Flux) of objects.

2-4. Accessing the Google Geocoder

Google provides a free geocoding web service that converts addresses into geographical coordinates. This exercise uses the WebClient to access the Google geocoder and converts the responses into Java objects.

2-5. Using the JDBC template

Spring provides a class called JdbcTemplate in the org.springframework.jdbc.core package. All it needs in order to work is a data source. It removes almost all the boilerplate code normally associated with JDBC. In this exercise, you'll use the JdbcTemplate to implement the standard CRUD (create, read, update, delete) methods on an entity.

2-6. Implementing the CRUD layer using JPA

The Java Persistence API (JPA) is a layer over the so-called persistence providers, the most common of which is Hibernate. With regular Spring, configuring JPA requires several beans, including an entity manager factory and a JPA vendor adapter. Fortunately, in Spring Boot, the presence of the JPA dependency causes the framework to implement all of that for you.

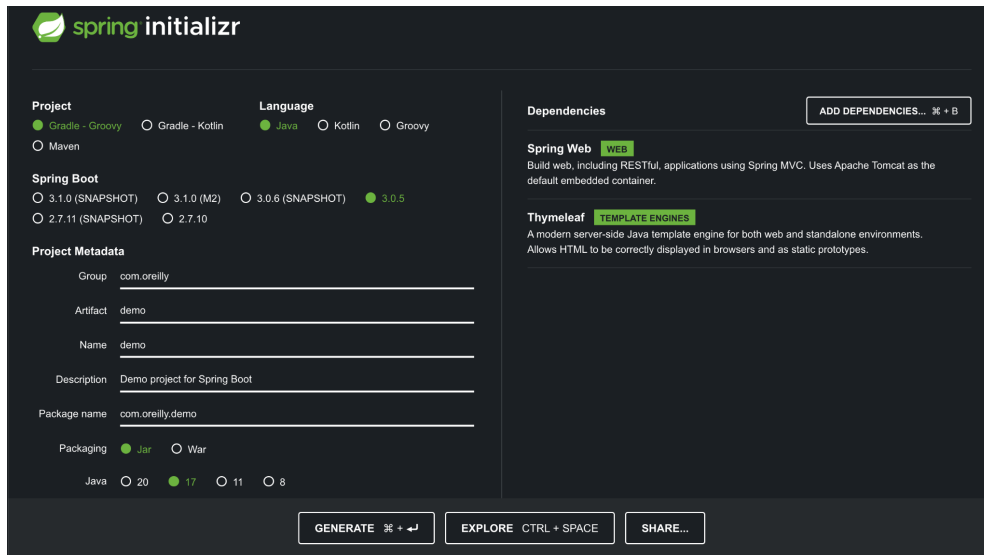
2-7. Using Spring Data

The Spring Data JPA project makes it incredibly easy to implement a DAO layer. You extend the proper interface, and the underlying infrastructure generates all the implementations for you.

2-1. Creating a New Project

a. Creating a New Spring Project with Gradle

1. Use <https://start.spring.io/> to create a new Spring Gradle Project.
 - *build.gradle* in Gradle===*pom.xml* in Maven
 - Add dependencies:
Spring Web and *Thymeleaf* dependencies



spring initializr

Project
☒ Gradle - Groovy ☐ Gradle - Kotlin ☐ Maven

Language
☒ Java ☐ Kotlin ☐ Groovy

Spring Boot
☐ 3.1.0 (SNAPSHOT) ☐ 3.1.0 (M2) ☐ 3.0.6 (SNAPSHOT) ☒ 3.0.5
☐ 2.7.11 (SNAPSHOT) ☐ 2.7.10

Project Metadata
 Group:
 Artifact:
 Name:
 Description:
 Package name:
 Packaging: ☒ Jar ☐ War
 Java: ☐ 20 ☒ 17 ☐ 11 ☐ 8

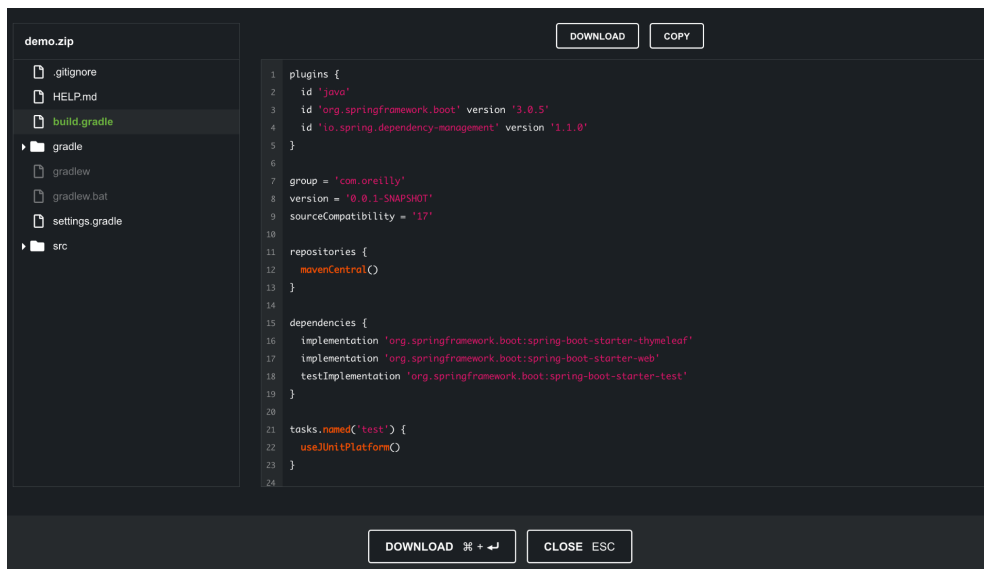
Dependencies ADD DEPENDENCIES... ⌘ + B

Spring Web WEB
 Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Thymeleaf TEMPLATE ENGINES
 A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

GENERATE ⌘ + ↵ EXPLORE CTRL + SPACE SHARE...

- Click on "Explore" to preview how the Gradle Project looks like:



demo.zip DOWNLOAD COPY

- .gitignore
- HELP.md
- build.gradle
- gradle
- gradlew
- gradlew.bat
- settings.gradle
- src

```

1 plugins {
2   id 'java'
3   id 'org.springframework.boot' version '3.0.5'
4   id 'io.spring.dependency-management' version '1.1.0'
5 }
6
7 group = 'com.oreilly'
8 version = '0.0.1-SNAPSHOT'
9 sourceCompatibility = '17'
10
11 repositories {
12   mavenCentral()
13 }
14
15 dependencies {
16   implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
17   implementation 'org.springframework.boot:spring-boot-starter-web'
18   testImplementation 'org.springframework.boot:spring-boot-starter-test'
19 }
20
21 tasks.named('test') {
22   useJUnitPlatform()
23 }
24
  
```

DOWNLOAD ⌘ + ↵ CLOSE ESC

- Download the Demo.zip
- Import the download.zip to IntelliJ/Eclipse or any other platform. Here I am using IntelliJ Community.

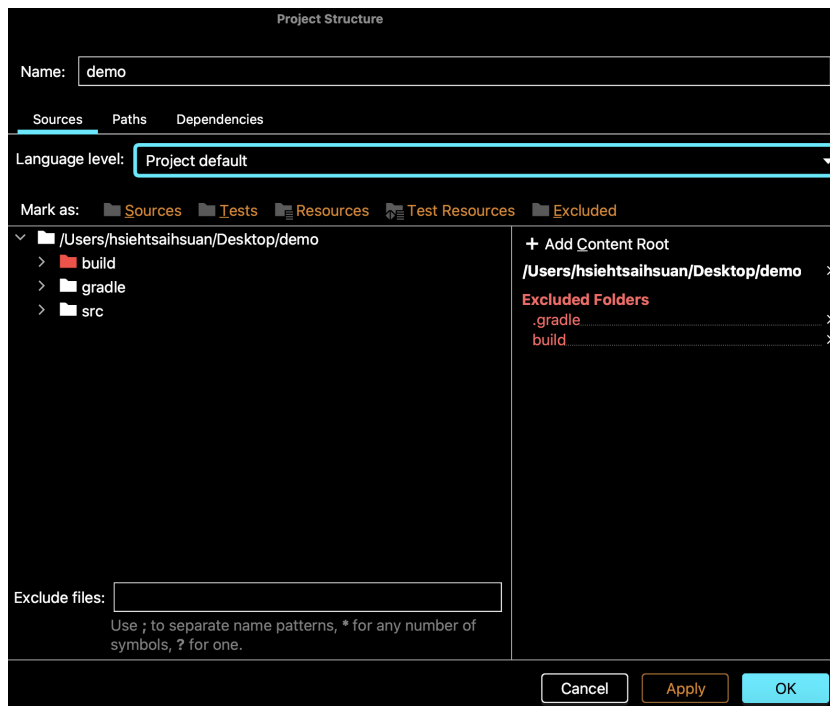
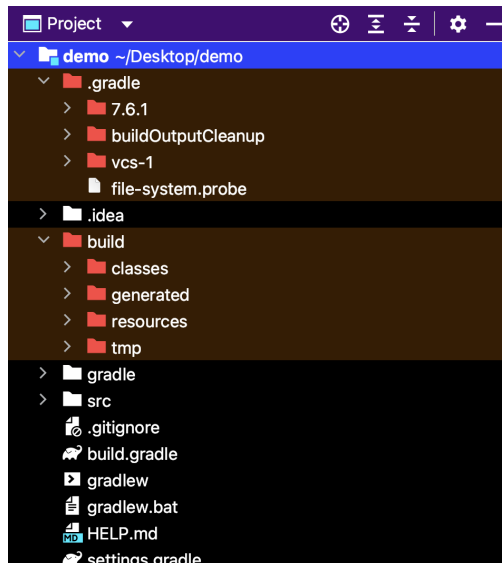
CONFUSION: Why are some files(.gradle and build) red? Is it presumably a bug?

ANS from ChatGPT:

No, there is nothing inherently wrong with excluded folders in a project structure being shown in red. In many integrated development environments (IDEs), red usually signifies that the folder or file is excluded from the current

build or execution path, and therefore won't be included in the final build artifact.

Excluding certain folders like ".gradle" and "build" is common practice in Java projects because these folders contain intermediate files and build artifacts that are generated during the build process and are not needed for the final application. Excluding them can help reduce clutter in the project structure and improve build performance.



DEBUG1:

- **Bug:** When running the main() method, the console shows the error *Unsupported class file major version*

- **Why the bug?**

Reason1: The "Unsupported class file major version 64" error typically occurs when you try to run a Java program that was compiled with a newer version of the Java compiler than the version of Java that is installed on your machine.

In this case, the "major version 64" indicates that the class file was compiled with Java 16, which is not supported by the version of Java that you are using.

Reason2: (My case here!)

If you are seeing the "Unsupported class file major version 64" error in IntelliJ IDEA despite using JDK 17, it could be because the project is configured to use an older version of the JDK.

- **Fix the bug:**

Open your project in IntelliJ IDEA.

Open the Project Structure dialog by pressing "**Command+;**" on Mac.

Under the "Project Settings" section on the left-hand side, click on "Project".

In the "Project SDK" dropdown, make sure that JDK 17 is selected. If it is not already installed, you can download and install it from the official Java website.

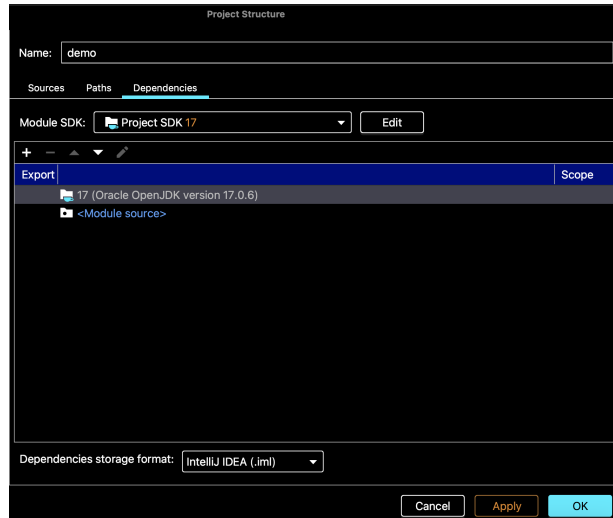
Under the "Project Settings" section on the left-hand side, click on "Modules".

In the "Sources" tab, make sure that the "Language level" dropdown is set to at least "17".

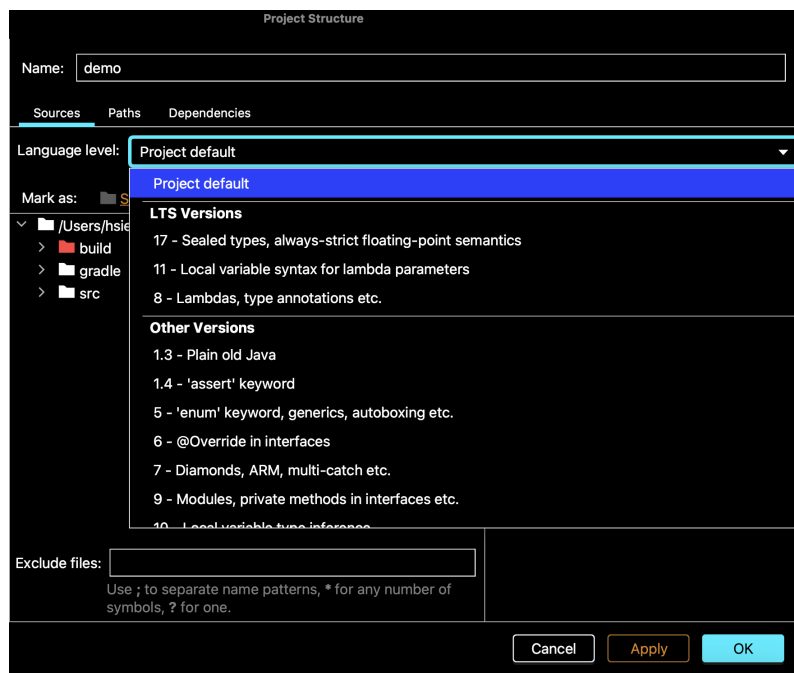
In the "Dependencies" tab, make sure that any external dependencies are also using a version of the JDK that is compatible with JDK 17.

Click "OK" to save your changes.

⇒ Check that in Project Structure, I am really using the JDK17



⇒ if not, select the correct one from Project Structure/Language Level:



DEBUG2:

- **Bug:** Execution failed for task ':DemoApplication.main()'.
 > Process 'command'
 '/Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home/bin/java'
 finished with non-zero exit value 1
- **Why the bug?**

Using the Debug in IntelliJ, it says that another Port is already using Port 8080. So that's why the Tomcat cannot be started properly.

- **Fix the bug:**

Go to Terminal, find the port and kill it.

Then rerun the DemoApplication main().

Is of -i :8080

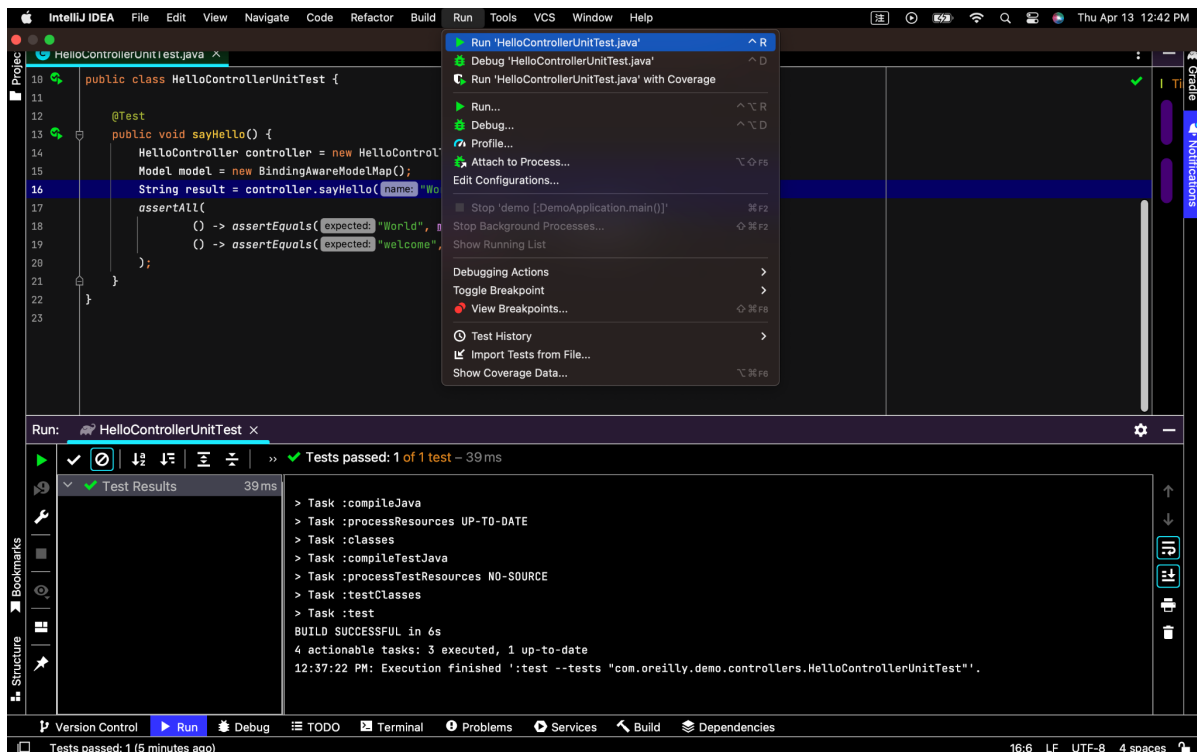
kill <PID>

(Replace <PID> with the actual PID of the process you want to stop.)

b. Add a HelloController.java in this project and build a test.java at Unit-Test-Level (here HelloControllerUnitTest.java) to test the controller's functionality.

(Because the controller is a simple POJO, you can unit test it by simply instantiating the controller and calling its sayHello method directly.)

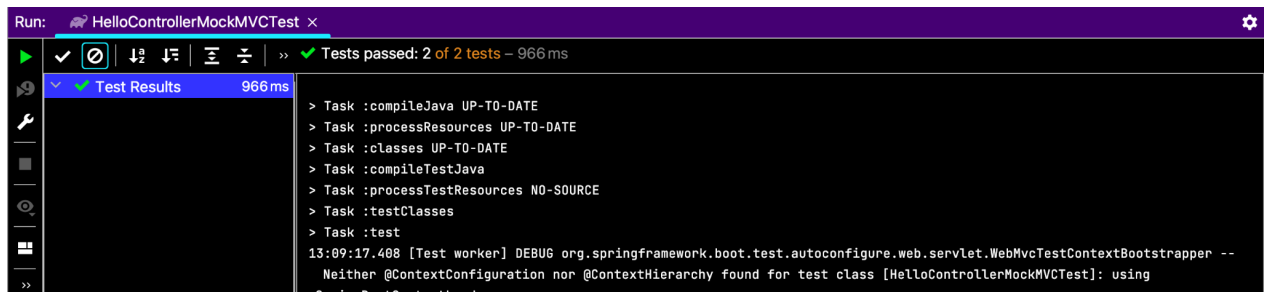
⇒ the test of "HelloControllerUnitTest.java" passed. This means our controller is functioning well so far.



- c. To perform an **integration test** instead, use the MockMVC classes available in Spring. Create a new class called **HelloControllerMockMVCTest**.

**This part of the exercise is using an “integration test” to show the main pros/cons of @WebMvcTest and @SpringBootTest.

The MockMVC Test passed, as shown below:



```
Run: HelloControllerMockMVCTest x
✓ Tests passed: 2 of 2 tests - 966 ms
Test Results 966 ms
> Task :compileJava UP-TO-DATE
> Task :processResources UP-TO-DATE
> Task :classes UP-TO-DATE
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses
> Task :test
13:09:17.408 [Test worker] DEBUG org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTestContextBootstrapper --
Neither @ContextConfiguration nor @ContextHierarchy found for test class [HelloControllerMockMVCTest]: using
SpringBootTestContextLoader
```

2-2. Add a Rest Controller

Goal of this Exercise: Add a HelloRestController.java and Greeting.java(a POJO).

What we are doing here is simply sending an HTTP request(by ourselves manually, we are hitting the URL by ourselves and pretend we're clients haha) to the URL "/rest" And the RestController handles this request with @RequestParam and returns a response with a new Greeting object(which is defined in another class, Greeting.java).

a. Terminology Clarification:

1. Q:What is REST?

A: **REST=Representational State Transfer**

In brief, The REST is an *architectural style* that is a set of principles that define how web services should be designed to be scalable, reliable, and easy to maintain. There are 5 principles: Client-Server architecture, Stateless, Cacheable, Layered System, Uniform interface.

2. Q:How is REST related to RESTful?

A: A RESTful web service is a web service that is designed and implemented using the REST architectural style.

3. Q: Chatgpt summarizes the **RestController.java** that "this class provides a simple way to (1)handle GET requests to a (2)RESTful endpoint in a (3)Spring Boot web application, and to (4)return a response in the form of a JSON object."

4.

****Q:Endpoint:** RESTful endpoint refers to a URI (Uniform Resource Identifier) that uniquely identifies a resource and the HTTP methods (such as GET, POST, PUT, DELETE, etc.) that can be used to interact with that resource.

****Q:What is a resource?** In the context of a RESTful API, a resource is an object or entity that can be accessed via a unique identifier or URL (Uniform Resource Locator).

RestController.java:

```
@RestController
public class HelloRestController {

    @GetMapping("/rest")
    public Greeting greet(@RequestParam(required = false,
        defaultValue = "World") String name) {
        return new Greeting(String.format("Hello, %s!", name));
    }
}
```

(1)handle GET requests ⇒ clue in @**Get**Mapping

(2)RESTful endpoint ⇒ clue in @**Rest**Controller & ("**/rest**")⇒ this one itself is an Endpoint. It has nothing to do with "rest". It just happens to be called "rest".

(3)Spring Boot web application ⇒ clue in @**Rest**Controller

(4)return a response in the form of a JSON object ⇒ clue in "the fact that it returns a Greeting object indicates that the response will be returned in the

form of a JSON object." ⇒ this doesn't make sense for me. How do you know Greeting is a JSON object?

The reason we know that the Greeting object will be returned as a JSON response is because of the **@RestController** annotation on the class. This annotation tells Spring that **this class is a RESTful controller that will handle HTTP requests and produce HTTP responses, and that the response should be serialized in the format specified by the Accept header of the HTTP request.**

When Spring receives a request with an Accept header of **application/json (which is the default for most web clients)**, it will automatically serialize the response object (in this case, the Greeting object) to JSON format before sending it back in the HTTP response body.

In other words, the @RestController annotation is what triggers the automatic serialization of the response object to JSON format. If we were to remove this annotation and replace it with @Controller, for example, we would need to add an additional annotation (@ResponseBody) to the greet() method to indicate that the method's return value should be serialized to JSON format.

2-3. Building a REST client

Goal of this Exercise:

This exercise uses the new **reactive web client(WebClient)** to access a RESTful web service. The template converts the response into an object for the rest of the system. Older Spring application, RestTemplate, is being gradually replaced with WebClient. **WebClient returns responses of type Mono and Flux.** They are "promises" that return a single object (for Mono) or a collection (for Flux) of objects.

****Terminology Clarification:**

****Q:What is the difference between Exercise 2-2 Building a REST controller and 2-3 Building a REST client?**

****A:** A REST client is a program or application that sends requests to a RESTful API, while a REST controller is a program or application that receives and processes requests from a RESTful API. In a typical RESTful architecture, the client and the server (which includes the RESTful API and the controller) communicate with each other over the HTTP protocol. The client sends requests to the server, which processes them using the RESTful API and the controller, and sends back responses to the client.

****Q:** What is Reactive Application?

****A:** Reactive applications are software applications that are designed to be more responsive, resilient, and scalable in the face of high levels of concurrency and varying workloads. Reactive programming is an approach to software development that emphasizes building systems that are responsive, resilient, and elastic.

Some popular frameworks and technologies for building reactive applications include **Spring WebFlux, Akka, Vert.x, and Reactor**. These frameworks provide tools and libraries that make it easier to build responsive, resilient, and scalable applications.

The screenshot shows an IDE with the following components:

- Project Explorer:** Shows the file structure: `restclient > src > test > java > com > oreilly > restclient > services > AstroServiceTest`.
- Editor:** Displays the code for `AstroServiceTest.java`. The code includes package declarations, imports for `AstroResponse`, `Test`, `Logger`, `LoggerFactory`, `Autowired`, `SpringBootTest`, `assertEquals`, and `assertTrue`. It also features the `@SpringBootTest` annotation and the start of the `AstroServiceTest` class.

```
1 package com.oreilly.restclient.services;
2
3 import com.oreilly.restclient.json.AstroResponse;
4 import org.junit.jupiter.api.Test;
5 import org.slf4j.Logger;
6 import org.slf4j.LoggerFactory;
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.boot.test.context.SpringBootTest;
9
10 import static org.junit.jupiter.api.Assertions.assertEquals;
11 import static org.junit.jupiter.api.Assertions.assertTrue;
12
13 @SpringBootTest
14 public class AstroServiceTest {
```
- Run Console:** Shows the execution results: `>> Tests passed: 1 of 1 test - 1sec 328ms`. Below this, a list of tasks is shown, all with a status of `UP-TO-DATE`:
 - `> Task :compileJava UP-TO-DATE`
 - `> Task :processResources UP-TO-DATE`
 - `> Task :classes UP-TO-DATE`
 - `> Task :compileTestJava UP-TO-DATE`
 - `> Task :processTestResources NO-SOURCE`
 - `> Task :testClasses UP-TO-DATE`
- Bottom Bar:** Contains icons for Version Control, Run, Debug, TODO, Problems, Terminal, Services, Build, and Dependencies.

Bug: TYPO!!! Response!!!

2-4. Accessing the Google Geocoder

Goal of this exercise:

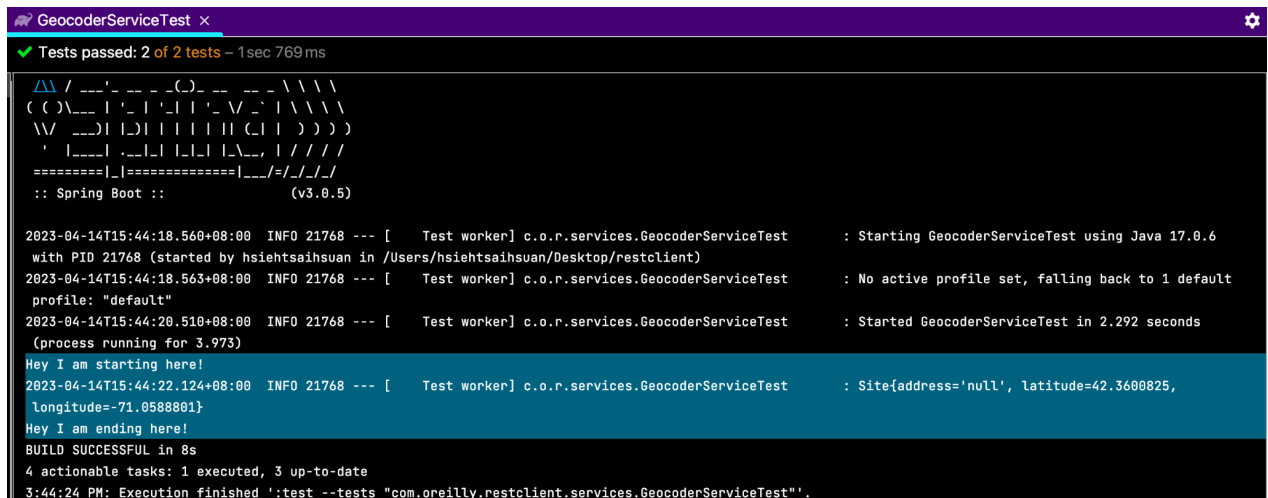
Google provides a free geocoding web service that converts addresses into geographical coordinates.

This exercise uses the `WebClient` to access the Google geocoder and converts the responses into Java objects.

logging message provided by slf4 are massive, the only thing I care here is if my testing codes are functioning well:

⇒ only the blue-backgrounded-part is the corresponding code in
GeocoderServiceTest:

```
@Test
public void getLatLngWithoutStreet() {
    System.out.println("Hey I am starting here!");
    Site site = service.getLatLng(...address: "Boston", "MA");
    logger.info(site.toString());
    assertEquals(
        () -> assertEquals(expected: 42.36, site.getLatitude(), delta: 0.01),
        () -> assertEquals(expected: -71.06, site.getLongitude(), delta: 0.01)
    );
    System.out.println("Hey I am ending here!");
}
```



```
GeocoderServiceTest x
✓ Tests passed: 2 of 2 tests - 1sec 769ms

Δ / _ _ _ ' _ _ _ ( _ _ _ _ _ \ \ \ \
( ( _ _ _ ' _ _ _ ' _ _ _ \ \ \ \
\ \ _ _ _ ) _ _ _ | | | | | ( _ _ _ ) ) )
' _ _ _ | _ _ _ | _ _ _ | _ _ _ | / / / /
=====|_|=====|_|_/ _ _ _ /
:: Spring Boot ::                (v3.0.5)

2023-04-14T15:44:18.560+08:00 INFO 21768 --- [    Test worker] c.o.r.services.GeocoderServiceTest : Starting GeocoderServiceTest using Java 17.0.6
with PID 21768 (started by hsiehtsaihuan in /Users/hsiehtsaihuan/Desktop/restclient)
2023-04-14T15:44:18.563+08:00 INFO 21768 --- [    Test worker] c.o.r.services.GeocoderServiceTest : No active profile set, falling back to 1 default
profile: "default"
2023-04-14T15:44:20.510+08:00 INFO 21768 --- [    Test worker] c.o.r.services.GeocoderServiceTest : Started GeocoderServiceTest in 2.292 seconds
(process running for 3.973)
Hey I am starting here!
2023-04-14T15:44:22.124+08:00 INFO 21768 --- [    Test worker] c.o.r.services.GeocoderServiceTest : Site{address='null', latitude=42.3600825,
longitude=-71.0588801}
Hey I am ending here!
BUILD SUCCESSFUL in 8s
4 actionable tasks: 1 executed, 3 up-to-date
3:44:24 PM: Execution finished 'test --tests "com.oreilly.restclient.services.GeocoderServiceTest"'
```

Even though the test passed, a further inspection showed that the testing
results are `Site{address='null', latitude=42.XXXX}`

Why? Because the address fields of the sites are null! That's because our
Result class has a String field called formattedAddress, but the JSON response
uses underscores instead of camel case (i.e., formatted_address).

Fixation:

One time fix: add an annotation to the `formatted_address` field in the `Result` class. The `@JsonProperty` annotation is a general purpose mechanism you can use whenever the property in the bean does not match the JSON field. Run your test again and see that the `name` value in the `Site` is now correct.

Global fix:

21. The other way to fix the issue is to set a global property that converts all camel case properties to underscores during the JSON parsing process. To use this, first remove the `@JsonProperty` annotation from `Result`.
22. We will then add the required property to a YAML properties file. By default, Spring Boot generates a file called `application.properties` in the `src/main/resources` folder. Rename that file to `application.yml`
23. Inside `application.yml`, add the following setting:

```
spring:
  jackson:
    property-naming-strategy: SNAKE_CASE
```

24. Once again run the tests and see that the `address` field in `Site` is set correctly. The advantage of the YAML file is that you can nest multiple properties without too much code duplication.

Now run the test again, it shows the Address!

```
GeocoderServiceTest x
✓ Tests passed: 2 of 2 tests – 1sec 518ms
.. Spring Boot .. (v3.0.5)

2023-04-14T15:59:38.044+08:00 INFO 31097 --- [ Test worker] c.o.r.services.GeocoderServiceTest : Starting GeocoderServiceTest using Java 17.0.6
with PID 31097 (started by hsiehsaihsuan in /Users/hsiehsaihsuan/Desktop/restclient)
2023-04-14T15:59:38.046+08:00 INFO 31097 --- [ Test worker] c.o.r.services.GeocoderServiceTest : No active profile set, falling back to 1 default
profile: "default"
2023-04-14T15:59:39.823+08:00 INFO 31097 --- [ Test worker] c.o.r.services.GeocoderServiceTest : Started GeocoderServiceTest in 2.062 seconds
(process running for 3.51)
Hey I am starting here!
2023-04-14T15:59:41.242+08:00 INFO 31097 --- [ Test worker] c.o.r.services.GeocoderServiceTest : Site{address='Boston, MA, USA', latitude=42
.3608825, longitude=-71.0588801}
Hey I am ending here!
BUILD SUCCESSFUL in 9s
4 actionable tasks: 2 executed, 2 up-to-date
```

2-5. Using the JDBC template

2-6. Implementing the CRUD layer using JPA

1. Add configuration in the “application.properties”:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

```
spring.h2.console.enabled=true
spring.h2.console.path=/db-console
```

2. Run a main() method.
3. Navigate to **http://localhost:8080/db-console** to access the H2 Database console.

A snapshot of the H2 Database.

← → ↻ 🏠 ⓘ localhost:8080/db-console/login.jsp?jsessionid=f90

🌐 AJAX

English ▾ Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded) ▾

Setting Name:

Driver Class:

JDBC URL:

User Name:

Password:

2-7. Using Spring Data

A snapshot of the HAL browser from Spring.

← → ↻ 🏠 ⓘ localhost:8080/explorer/index.html#uri=/

🌐 AJAX Other Bookmarks

HAL Explorer

Edit Headers

Links

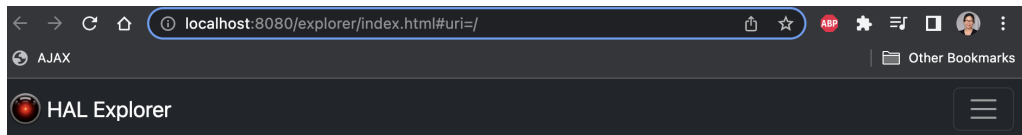
| Relation | Name | Title | HTTP Request | Doc |
|----------|------|-------|--|-----|
| officers | | | ⏪ ⏩ ⏴ ⏵ ✖ | |
| profile | | | ⏪ ⏩ ⏴ ⏵ ✖ | |

Response Status

200 (OK)

Response Headers

| | |
|-------------------|---|
| connection | keep-alive |
| content-type | application/hal+json |
| date | Sun, 16 Apr 2023 07:41:14 GMT |
| keep-alive | timeout=60 |
| transfer-encoding | chunked |
| vary | Origin, Access-Control-Request-Method, Access-Control-Request-Headers |













Edit Headers

/

Go!

Links

| Relation | Name | Title | HTTP Request | Doc |
|----------|------|-------|---|-----|
| officers | | |      | |
| profile | | |      | |

Response Status

200 (OK)

Response Headers

| | |
|-------------------|-------------------------------|
| connection | keep-alive |
| content-type | application/hal+json |
| date | Sun, 16 Apr 2023 07:41:14 GMT |
| keep-alive | timeout=60 |
| transfer-encoding | chunked |