

Table of Contents

Overview	3
Create Simple Rest API	4
1. Prepare Spring Boot Project.....	4
2. Create Object Model	6
3. Create Service Process	6
4. Create Jersey Configuration Class	7
5. Setup Server Port and Run the application	7
Implementing Asynchronous Methods	9
1. Prepare Spring Boot Project.....	12
2. Enable Async Method in Entry Point Spring Boot Application	13
3. Prepare Model Object Mapping.....	14
4. Prepare the Service Class for implementing Asynchronous Logic.....	15
5. Prepare Runner Class	17
6. Run the App	18
Schedule Task.....	20
1. Create Spring Boot Project	20
2. Enabling Scheduler in Entry Point for Spring Boot Application	22
3. Configure the Interval timing with type FixedRate.....	22
4. Run and test the Application.....	24
5. Update the interval execution type with fixed Delay	25
6. Update the interval execution type with fixed Rate and initial Delay.....	27
7. Update the interval execution type with Fixed Rate String.....	29
Scheduling Asynchronous Tasks.....	30
1. Create Spring Boot Project	30
2. Enable Synchronous and Scheduler in Entry Point Spring Boot Application	32
3. Prepare Model Object	33
4. Prepare Business Logic	34
5. Prepare Controller.....	35
6. Run And Test Spring Boot Asynchronous Scheduled Application.....	37
7. Update And Increase thread Schedule	38
Using Request Parameters and Dynamic Paths	39

Spring Boot Microservices: Asynchronous Methods, Schedulers, & Forms

1. Create New Spring Boot Project.....	39
2. Entry Point Spring Boot Application	41
3. Prepare Controller Object.....	41
4. Accepting Request Parameter name as Variable Input	44
5. Accepting Path Variable as Variable Input	45
6. Accepting Form input	46
Using Form	48
7. Inspect Maven Dependency in pom.xml	48
8. Inspect Entry Point of Spring Boot Application	49
9. Create Form Model Object	50
10. Create Controller	51
11. Create View Page	53
12. Run and Test The Application.....	56
Performing Form Validation	57
13. Add new dependency for Validation in POM.xml.....	58
14. Include Validation on each property in Model Object	59
15. Update routing Controller when handling invalid validation	61
Course Summary	66
Quiz.....	67

Overview

Spring Boot is an open source Java-based framework used to create microservices. Spring Boot makes it easy to create standalone production grade spring-based applications that just work.

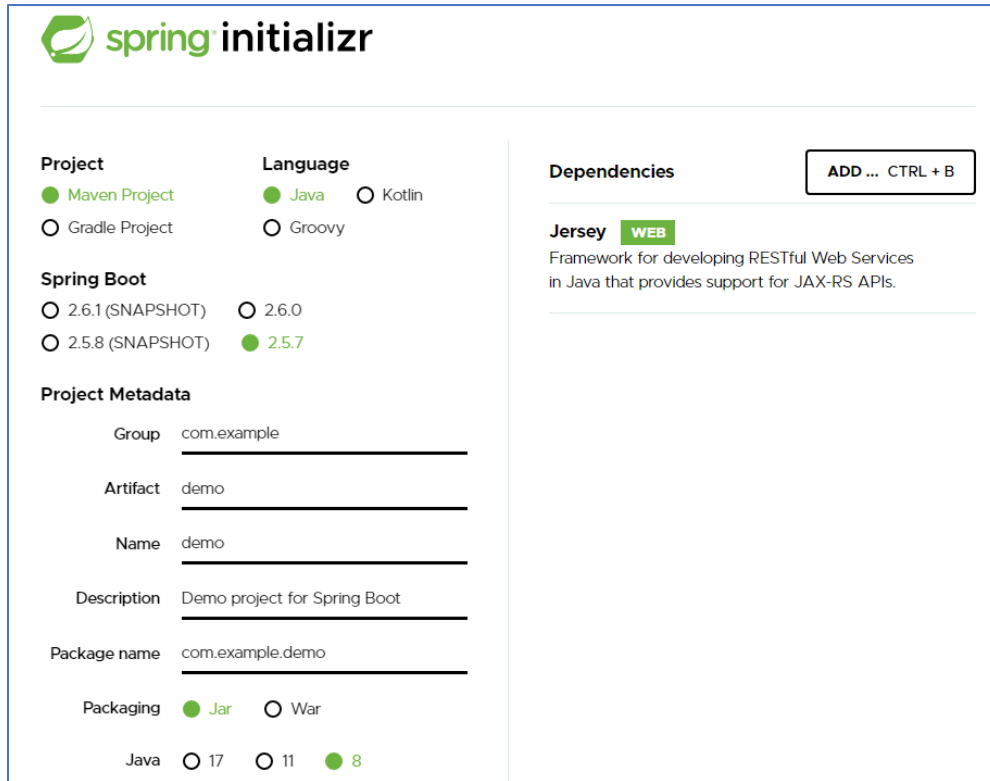
Spring Boot can be thought of as an extension of the spring framework. Which is used to bring diverse components together to build enterprise applications. Even a simple spring applications may involve multiple dependencies. And managing these dependencies along with their versions can be quite complex. Spring Boot is an opinionated framework. Which makes building applications easy by providing sensible defaults for most of the libraries that you need. Yet, allowing you the flexibility of configuring specific classes. Having understood how we get started with the Spring Boot application, we will now explore some interesting features that Spring Boot has to offer us. We'll see how easy it is to run asynchronous tasks using Spring Boot. All it needs is a few annotations set up correctly. We'll configure our applications to run scheduled tasks at a fixed rate and fixed delay. We'll also set up forms using the Thymeleaf template engine and perform built in validation on the form fields.

Once you're done with this course, you'll be able to configure handler mappings in spring MVC for request paths. Extract request parameters and path variables, run asynchronous and scheduled operations in Spring Boot. And finally perform validation of form input that a user specifies.

Create Simple Rest API

1. Prepare Spring Boot Project

Generate template from <http://start.spring.io>, extract it and open this maven project in eclipse



The screenshot shows the Spring Initializr web application interface. It is divided into several sections for configuring a new Spring Boot project:

- Project:** Includes radio buttons for **Maven Project** (selected) and **Gradle Project**.
- Language:** Includes radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for versions **2.6.1 (SNAPSHOT)**, **2.6.0**, **2.5.8 (SNAPSHOT)**, and **2.5.7** (selected).
- Project Metadata:** A form with input fields for **Group** (com.example), **Artifact** (demo), **Name** (demo), **Description** (Demo project for Spring Boot), and **Package name** (com.example.demo).
- Packaging:** Includes radio buttons for **Jar** (selected) and **War**.
- Java:** Includes radio buttons for versions **17**, **11**, and **8** (selected).
- Dependencies:** A section with an **ADD ... CTRL + B** button and a list of dependencies. Currently, **Jersey** is selected with a **WEB** tag. A description for Jersey is provided: "Framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs."

Spring Boot Jersey

Spring Boot Jersey tutorial shows how to set up a simple RESTful application with Jersey in a Spring Boot application. Jersey is an alternative to Spring RESTful applications created with `@RestController`.

Spring is a popular Java application framework for creating enterprise applications. Spring Boot is the next step in evolution of Spring framework. It helps create stand-alone, production-grade Spring based applications with minimal effort. It promotes using the convention over configuration principle over XML configurations.

RESTful application

A RESTful application follows the REST architectural style, which is used for designing networked applications. RESTful applications generate HTTP requests performing CRUD (Create/Read/Update/Delete) operations on resources. RESTful applications typically return data in JSON or XML format.

JAX-RS

Java API for RESTful Web Services (JAX-RS) is a Java programming language API specification that provides support in creating web services according to the Representational State Transfer

(REST) architectural pattern. JAX-RS uses annotations to simplify the development and deployment of web service clients and endpoints. JAX-RS is an official part of Java EE.

Jersey

Jersey is an open source framework for developing RESTful Web Services in Java. It is a reference implementation of the Java API for RESTful Web Services (JAX-RS) specification.

The following application is a simple Spring Boot RESTful application created with Jersey.

pom.xml



```
demo/pom.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.5.7</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.example</groupId>
12  <artifactId>demo</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>demo</name>
15  <description>Demo project for Spring Boot</description>
16  <properties>
17    <java.version>1.8</java.version>
18  </properties>
19  <dependencies>
20    <dependency>
21      <groupId>org.springframework.boot</groupId>
22      <artifactId>spring-boot-starter-jersey</artifactId>
23    </dependency>
24
25    <dependency>
26      <groupId>org.springframework.boot</groupId>
27      <artifactId>spring-boot-starter-test</artifactId>
28      <scope>test</scope>
29    </dependency>
30  </dependencies>
31
32  <build>
33    <plugins>
34      <plugin>
35        <groupId>org.springframework.boot</groupId>
36        <artifactId>spring-boot-maven-plugin</artifactId>
37      </plugin>
38    </plugins>
39  </build>
40
41 </project>
```

This is the Maven build file. Spring Boot starters are a set of convenient dependency descriptors which greatly simplify Maven configuration. The spring-boot-starter-parent has some common configurations for a Spring Boot application. The spring-boot-starter-jersey is a starter for building RESTful web applications using JAX-RS and Jersey. It is an alternative to spring-boot-starter-web. The spring-boot-starter-test is a starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito.

2. Create Object Model

User.java

```

1 package com.example.demo;
2
3 public class User {
4
5     private String name;
6     private String blog;
7     private String type;
8     private String url;
9
10
11     public User(String name, String blog, String type, String url) {
12         this.name = name;
13         this.blog = blog;
14         this.type = type;
15         this.url = url;
16     }
17
18     public String getName() {
19         return name;
20     }
21     public void setName(String name) {
22         this.name = name;
23     }
24     public String getBlog() {
25         return blog;
26     }
27     public void setBlog(String blog) {
28         this.blog = blog;
29     }
30     public String getType() {
31         return type;
32     }
33     public void setType(String type) {
34         this.type = type;
35     }
36     public String getUrl() {
37         return url;
38     }
39     public void setUrl(String url) {
40         this.url = url;
41     }

```

3. Create Service Process

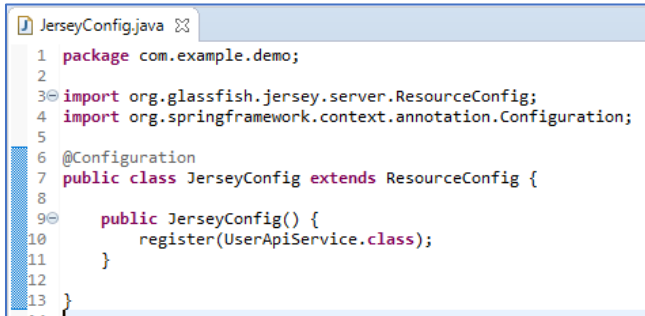
```

1 package com.example.demo;
2
3 import java.util.Random;
4
5 import javax.ws.rs.GET;
6 import javax.ws.rs.Path;
7 import javax.ws.rs.PathParam;
8 import javax.ws.rs.Produces;
9 import javax.ws.rs.core.MediaType;
10
11 import org.springframework.stereotype.Service;
12
13 @Service
14 @Path("/user")
15 public class UserApiService {
16
17     @GET
18     @Path("/{user}")
19     @Produces(MediaType.APPLICATION_JSON)
20     public User getUser(@PathParam("user") String name) {
21         System.out.println("get employee of " + name);
22
23         String userName = String.format("%s%s", name.substring(0, 1).toUpperCase(), name.substring(1).toLowerCase());
24         String blog = String.format("https://www.%s.com", name.toLowerCase());
25         String type = (new Random().nextInt(99999999) % 2 == 0) ? "Organization" : "User";
26         String url = String.format("https://api.github.com/users/%s", name.toLowerCase());
27
28         return new User(userName, blog, type, url);
29     }
30
31 }

```

4. Create Jersey Configuration Class

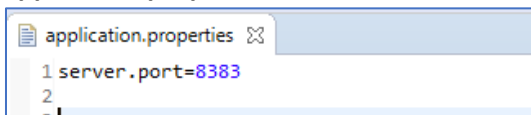
JerseyConfiguration.java

A screenshot of an IDE showing the code for JerseyConfiguration.java. The code is as follows:

```
1 package com.example.demo;
2
3 import org.glassfish.jersey.server.ResourceConfig;
4 import org.springframework.context.annotation.Configuration;
5
6 @Configuration
7 public class JerseyConfig extends ResourceConfig {
8
9     public JerseyConfig() {
10         register(UserApiService.class);
11     }
12
13 }
```

5. Setup Server Port and Run the application

application.properties

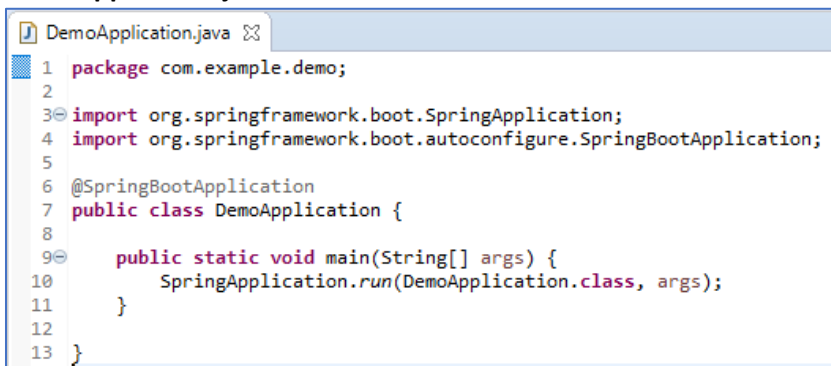
A screenshot of an IDE showing the application.properties file with the following content:

```
1 server.port=8383
2
```

Run the application

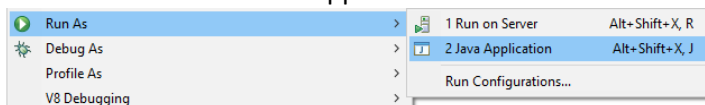
right click on Entry Point of Spring Boot Application (“DemoApplication.java”)

DemoApplication.java

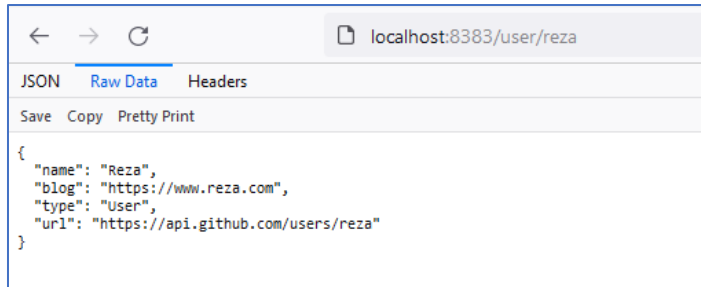
A screenshot of an IDE showing the code for DemoApplication.java. The code is as follows:

```
1 package com.example.demo;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class DemoApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(DemoApplication.class, args);
11     }
12
13 }
```

and select “Run As > Java Application”



open browser and hit localhost:8383/user/spring_user



Implementing Asynchronous Methods

In this demo, we'll see how we can make an asynchronous background request within our Spring Boot application. When you're building a complex enterprise grade application. There might be several expensive processes that you want to run in the background asynchronously. In this demo, we'll see how you can use futures within Spring Boot to run asynchronous operations.

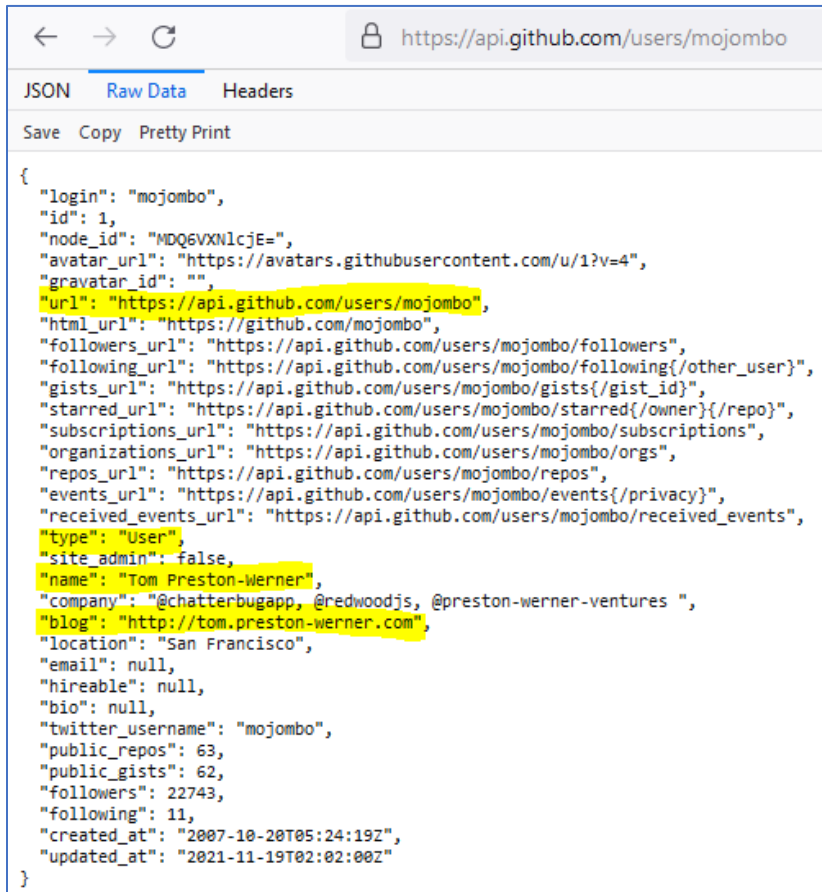
Now, we'll make asynchronous request to `api.github.com/users`.



```
[
  {
    "login": "mojombo",
    "id": 1,
    "node_id": "MDQ6VXNlcjE=",
    "avatar_url": "https://avatars.githubusercontent.com/u/1?v=4",
    "gravatar_id": "",
    "url": "https://api.github.com/users/mojombo",
    "html_url": "https://github.com/mojombo",
    "followers_url": "https://api.github.com/users/mojombo/followers",
    "following_url": "https://api.github.com/users/mojombo/following{/other_user}",
    "gists_url": "https://api.github.com/users/mojombo/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/mojombo/starred{/owner}/{/repo}",
    "subscriptions_url": "https://api.github.com/users/mojombo/subscriptions",
    "organizations_url": "https://api.github.com/users/mojombo/orgs",
    "repos_url": "https://api.github.com/users/mojombo/repos",
    "events_url": "https://api.github.com/users/mojombo/events{/privacy}",
    "received_events_url": "https://api.github.com/users/mojombo/received_events",
    "type": "User",
    "site_admin": false
  },
  {
    "login": "defunkt",
    "id": 2,
    "node_id": "MDQ6VXNlcjI=",
    "avatar_url": "https://avatars.githubusercontent.com/u/2?v=4",
    "gravatar_id": "",
    "url": "https://api.github.com/users/defunkt",
    "html_url": "https://github.com/defunkt",
    "followers_url": "https://api.github.com/users/defunkt/followers",
    "following_url": "https://api.github.com/users/defunkt/following{/other_user}",
    "gists_url": "https://api.github.com/users/defunkt/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/defunkt/starred{/owner}/{/repo}",
    "subscriptions_url": "https://api.github.com/users/defunkt/subscriptions",
    "organizations_url": "https://api.github.com/users/defunkt/orgs",
    "repos_url": "https://api.github.com/users/defunkt/repos",
    "events_url": "https://api.github.com/users/defunkt/events{/privacy}",
    "received_events_url": "https://api.github.com/users/defunkt/received_events",
    "type": "User",
    "site_admin": false
  },
  {
    "login": "pjhyett",
    "id": 3,
    "node_id": "MDQ6VXNlcjM=",
    "avatar_url": "https://avatars.githubusercontent.com/u/3?v=4",
    "gravatar_id": "",
    "url": "https://api.github.com/users/pjhyett",
    "html_url": "https://github.com/pjhyett",
    "followers_url": "https://api.github.com/users/pjhyett/followers",
    "following_url": "https://api.github.com/users/pjhyett/following{/other_user}",
    "gists_url": "https://api.github.com/users/pjhyett/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/pjhyett/starred{/owner}/{/repo}",
    "subscriptions_url": "https://api.github.com/users/pjhyett/subscriptions",
    "organizations_url": "https://api.github.com/users/pjhyett/orgs",
    "repos_url": "https://api.github.com/users/pjhyett/repos",
    "events_url": "https://api.github.com/users/pjhyett/events{/privacy}",
    "received_events_url": "https://api.github.com/users/pjhyett/received_events",
    "type": "User",
    "site_admin": false
  }
],
```

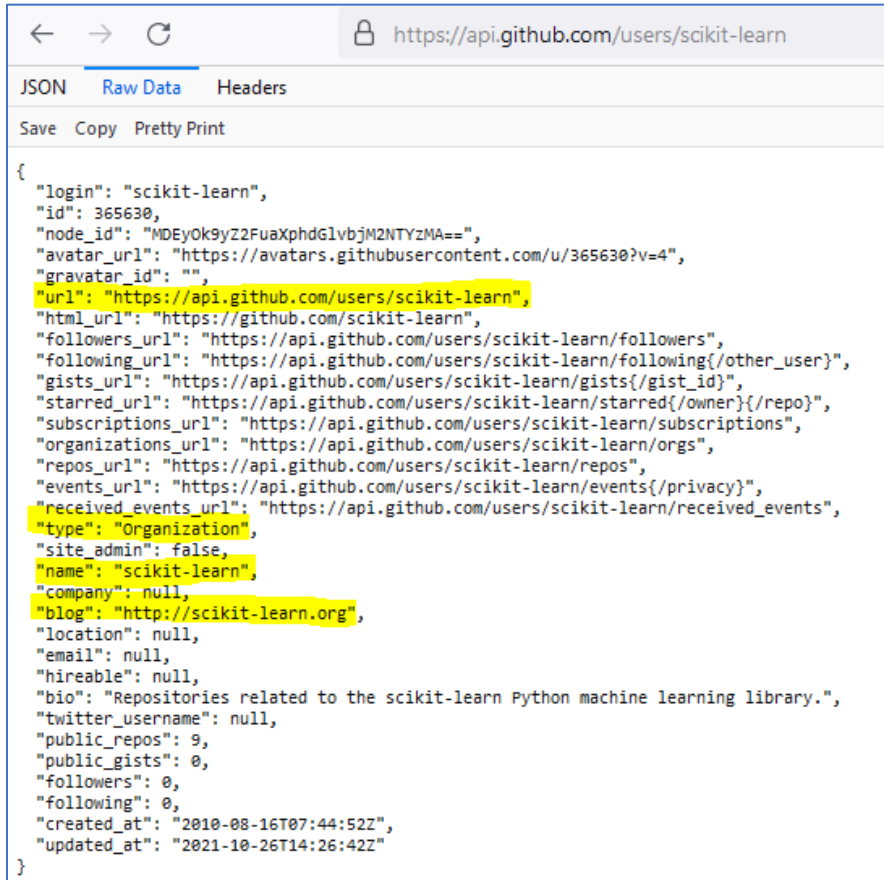
Let's take a look at some of the URLs that we'll be hitting in our demo.

I've made an http get request to the one of users page, and I get in the response some JSON. In the application that we'll be building, we're interested in four specific properties for this user. The name of the user, the blog, the type and the URL.



```
{
  "login": "mojombo",
  "id": 1,
  "node_id": "MDQ6VXNlcjE=",
  "avatar_url": "https://avatars.githubusercontent.com/u/1?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/mojombo",
  "html_url": "https://github.com/mojombo",
  "followers_url": "https://api.github.com/users/mojombo/followers",
  "following_url": "https://api.github.com/users/mojombo/following{/other_user}",
  "gists_url": "https://api.github.com/users/mojombo/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/mojombo/starred{/owner}/{/repo}",
  "subscriptions_url": "https://api.github.com/users/mojombo/subscriptions",
  "organizations_url": "https://api.github.com/users/mojombo/orgs",
  "repos_url": "https://api.github.com/users/mojombo/repos",
  "events_url": "https://api.github.com/users/mojombo/events{/privacy}",
  "received_events_url": "https://api.github.com/users/mojombo/received_events",
  "type": "User",
  "site_admin": false,
  "name": "Tom Preston-Werner",
  "company": "@chatterbugapp, @redwoodjs, @preston-werner-ventures ",
  "blog": "http://tom.preston-werner.com",
  "location": "San Francisco",
  "email": null,
  "hireable": null,
  "bio": null,
  "twitter_username": "mojombo",
  "public_repos": 63,
  "public_gists": 62,
  "followers": 22743,
  "following": 11,
  "created_at": "2007-10-20T05:24:19Z",
  "updated_at": "2021-11-19T02:02:00Z"
}
```

Let's make a request to another user. This time I'll do it within a new tab, the user is

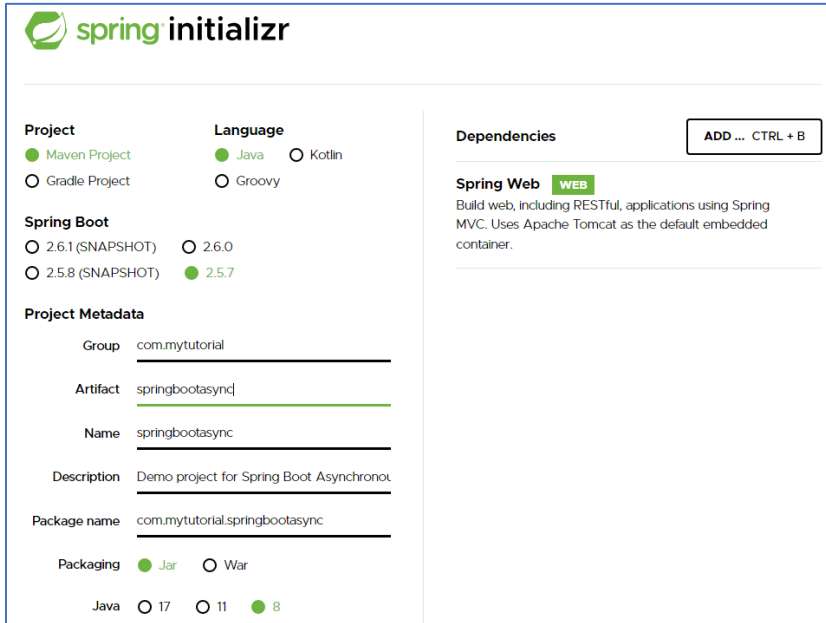


```
{
  "login": "scikit-learn",
  "id": 365630,
  "node_id": "MDEyOk9yZ2FuaxphdGlvbWJ2NTYzMA==",
  "avatar_url": "https://avatars.githubusercontent.com/u/365630?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/scikit-learn",
  "html_url": "https://github.com/scikit-learn",
  "followers_url": "https://api.github.com/users/scikit-learn/followers",
  "following_url": "https://api.github.com/users/scikit-learn/following{/other_user}",
  "gists_url": "https://api.github.com/users/scikit-learn/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/scikit-learn/starred{/owner}/{/repo}",
  "subscriptions_url": "https://api.github.com/users/scikit-learn/subscriptions",
  "organizations_url": "https://api.github.com/users/scikit-learn/orgs",
  "repos_url": "https://api.github.com/users/scikit-learn/repos",
  "events_url": "https://api.github.com/users/scikit-learn/events{/privacy}",
  "received_events_url": "https://api.github.com/users/scikit-learn/received_events",
  "type": "Organization",
  "site_admin": false,
  "name": "scikit-learn",
  "company": null,
  "blog": "http://scikit-learn.org",
  "location": null,
  "email": null,
  "hireable": null,
  "bio": "Repositories related to the scikit-learn Python machine learning library.",
  "twitter_username": null,
  "public_repos": 9,
  "public_gists": 0,
  "followers": 0,
  "following": 0,
  "created_at": "2010-08-16T07:44:52Z",
  "updated_at": "2021-10-26T14:26:42Z"
}
```

This references the scikit-learn project. Once again, we'll extract the *name*, *blog*, *type* and *URL* from this JSON response.

1. Prepare Spring Boot Project

Generate template from <http://start.spring.io>, extract it and open this maven project in eclipse



The Spring Initializr form is configured as follows:

- Project:** ☒ Maven Project, ☐ Gradle Project
- Language:** ☒ Java, ☐ Kotlin, ☐ Groovy
- Spring Boot:** ☐ 2.6.1 (SNAPSHOT), ☐ 2.6.0, ☐ 2.5.8 (SNAPSHOT), ☒ 2.5.7
- Project Metadata:**
 - Group: com.mytutorial
 - Artifact: springbootasync
 - Name: springbootasync
 - Description: Demo project for Spring Boot Asynchronous
 - Package name: com.mytutorial.springbootasync
- Packaging:** ☒ Jar, ☐ War
- Java:** ☐ 17, ☐ 11, ☒ 8
- Dependencies:** Spring Web (WEB) - Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

We'll now switch over to our Eclipse IDE and take a look at the dependencies that specified in **pom.xml**. We'll use the **spring-boot-starter-web** descriptor.

```

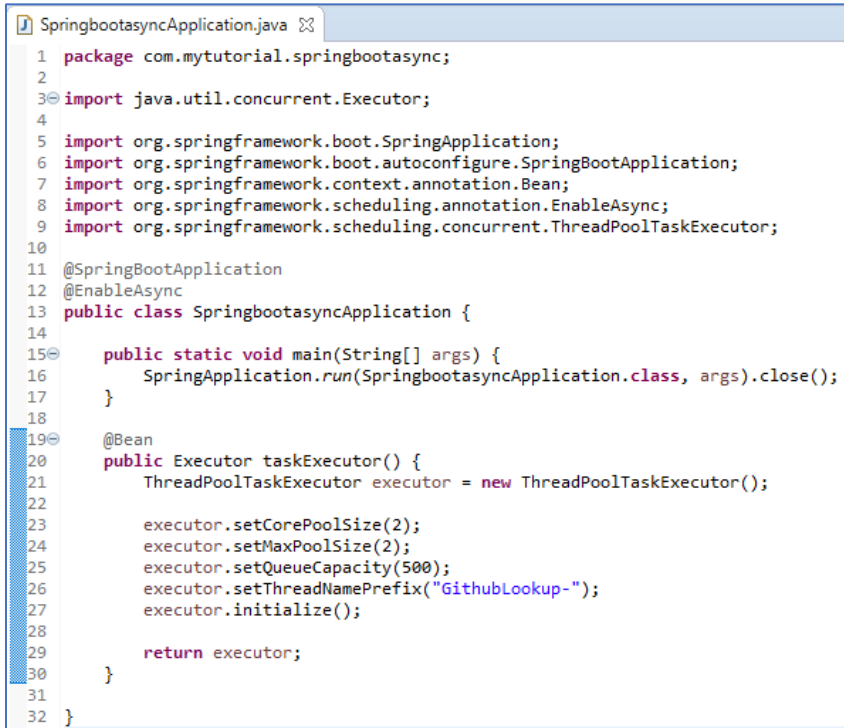
springbootasync/pom.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.5.7</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.mytutorial</groupId>
12  <artifactId>springbootasync</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>springbootasync</name>
15  <description>Demo project for Spring Boot Asynchronous Method </description>
16  <properties>
17    <java.version>1.8</java.version>
18  </properties>
19  <dependencies>
20    <dependency>
21      <groupId>org.springframework.boot</groupId>
22      <artifactId>spring-boot-starter-web</artifactId>
23    </dependency>
24
25    <dependency>
26      <groupId>org.springframework.boot</groupId>
27      <artifactId>spring-boot-starter-test</artifactId>
28      <scope>test</scope>
29    </dependency>
30  </dependencies>
31
32  <build>
33    <plugins>
34      <plugin>
35        <groupId>org.springframework.boot</groupId>
36        <artifactId>spring-boot-maven-plugin</artifactId>
37      </plugin>
38    </plugins>
39  </build>
40
41 </project>

```

2. Enable Async Method in Entry Point Spring Boot Application

The starter web template gives us access to a rest template that we can use to make URL requests. Let's now head over to our main application.java file,

SpringbootasyncApplication.java.



```

1 package com.mytutorial.springbootasync;
2
3 import java.util.concurrent.Executor;
4
5 import org.springframework.boot.SpringApplication;
6 import org.springframework.boot.autoconfigure.SpringBootApplication;
7 import org.springframework.context.annotation.Bean;
8 import org.springframework.scheduling.annotation.EnableAsync;
9 import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
10
11 @SpringBootApplication
12 @EnableAsync
13 public class SpringbootasyncApplication {
14
15     public static void main(String[] args) {
16         SpringApplication.run(SpringbootasyncApplication.class, args).close();
17     }
18
19     @Bean
20     public Executor taskExecutor() {
21         ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
22
23         executor.setCorePoolSize(2);
24         executor.setMaxPoolSize(2);
25         executor.setQueueCapacity(500);
26         executor.setThreadNamePrefix("GithubLookup-");
27         executor.initialize();
28
29         return executor;
30     }
31 }
32 
```

Notice that it tagged using the **@SpringBootApplication** annotation, that one we're familiar with. I've also used the **@EnableAsync** annotation.

The **@EnableAsync** annotation turns on Spring's ability to run asynchronous methods in a background thread pool. Rather than instantiating threads directly, this **@EnableAsync** uses an executor. And you can specify the executor that it should use using the task executor method, which I've done on line 20. Notice that I've annotated it as a bean that can be injected.

The name of this method here specified on line 20, **taskExecutor** is important. That's because the **@EnableAsync** annotation specifically looks for a method with this name. Another detail to note here is within our main method.

Notice we call **SpringApplication.run**, and as soon as the application will complete its execution, it'll close. That's what the **.close()** here tells us.

Within this **TaskExecutor** method, we instantiate a new **ThreadPoolTaskExecutor**.

And we set its core and max pool size to 2, which means at any point in time we can have two threads running simultaneously.

We invoke **executor.initialize** on line 27 and return an instance of this executor.

If you don't specify a task executor, **EnableAsync** will use one by default, a simple task executor.

3. Prepare Model Object Mapping

We've seen that when we make a request to a specific user's GitHub page, the response that is returned is in the JSON format. We need a way to convert this JSON response to an actual Java object. We'll define this user class here which represents the response from a specific GitHub user. Spring offers a number of different ways to convert a JSON response to a Java object representation.

User.java

```

1 package com.mytutorial.springbootasync.model;
2
3 import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
4
5 @JsonIgnoreProperties(ignoreUnknown = true)
6 public class User {
7
8     private String name;
9     private String blog;
10    private String type;
11    private String url;
12
13    public String getName() {
14        return name;
15    }
16    public void setName(String name) {
17        this.name = name;
18    }
19    public String getBlog() {
20        return blog;
21    }
22    public void setBlog(String blog) {
23        this.blog = blog;
24    }
25    public String getType() {
26        return type;
27    }
28    public void setType(String type) {
29        this.type = type;
30    }
31    public String getUrl() {
32        return url;
33    }
34    public void setUrl(String url) {
35        this.url = url;
36    }
37
38    @Override
39    public String toString() {
40        return String.format(
41            "User [name=%s, blog=%s, type=%s, url=%s ]",
42            name, blog, type, url);
43    }
44
45 }

```

By default, Spring Boot uses the **Jackson library**.

Notice that my user class has a specific annotation **@JsonIgnoreProperties**.

This indicates to spring that the JSON response that we get from the URL will contain properties. Over and above the properties that we are interested in and it should simply ignore those. The only properties that we're interested in is the *name* of the user, the *blog*, the *type* and the *URL*.

The member variables that you specify for this user object should match the JSON keys in the web response.

Spring will then use reflection to access the names of these member variables. And look up the corresponding JSON properties in the web response.

The remaining bits of code here are simply getters and setters for all of these member variables. At the very bottom, I've overridden the **toString** method of the object base class to print out the details of a single user.

4. Prepare the Service Class for implementing Asynchronous Logic

The actual asynchronous method call made to look up the GitHub users information will be done within this LookupService. This LookupService is a class annotated using the **@Service** annotation.

LookupService.java

```

1 package com.mytutorial.springbootasync.service;
2
3 import java.util.concurrent.CompletableFuture;
4
5 import org.slf4j.Logger;
6 import org.slf4j.LoggerFactory;
7 import org.springframework.boot.web.client.RestTemplateBuilder;
8 import org.springframework.scheduling.annotation.Async;
9 import org.springframework.stereotype.Service;
10 import org.springframework.web.client.RestTemplate;
11
12 import com.mytutorial.springbootasync.model.User;
13
14 @Service
15 public class LookupService {
16
17     private static final Logger logger = LoggerFactory.getLogger(LookupService.class);
18
19     private static final String GIT_HUB_USERS_URL = "https://api.github.com/users/%s";
20
21     private final RestTemplate restTemplate;
22
23     public LookupService(RestTemplateBuilder restTemplateBuilder) {
24         this.restTemplate = restTemplateBuilder.build();
25     }
26
27     @Async
28     public CompletableFuture<User> findUser(String user) throws InterruptedException {
29         logger.info(String.format("Looking up %s", user));
30         String url = String.format(GIT_HUB_USERS_URL, user);
31         User result = restTemplate.getForObject(url, User.class);
32         Thread.sleep(4000L);
33         return CompletableFuture.completedFuture(result);
34     }
35
36 }

```

This **@Service** annotation indicates that this is a Spring managed component. A Spring bean whose life cycle is taken care of by the Spring framework. The fact that it's **@Service** indicates intent, it tells us that this particular class holds business logic. This LookupService here will perform a lookup operation, looking up a particular GitHub user.

Now, the URL that we want to hit is specified in `GITHUB_USERS_URL` on line 19, *api.github.com/users/%s*.

We'll perform the actual lookup for a specific user using Spring's `restTemplate`.

This rest template is injected into the constructor of our `LookupService` on line 23.

The most important bit of code here in this `LookupService` is this `findUser` method. The first thing you should observe about this method is that it is annotated using the **@Async** annotation.

This **@Async** annotation is what tells our Spring Boot application that *the code for this method needs to be run on a separate thread*. It should not be on the main thread, it should be on a background thread.

You can see that the return value for this `find user` method is a completable future which holds a user object.

If you've worked with asynchronous programming in Java, you know that returning a future is a requirement for any async operation. This completable future will hold the user object representing our GitHub user.

Once the asynchronous method invocation is complete and the result is available. The input argument to this `FindUser` method is the user name in the form of a string. We then generate the URL link for this GitHub Users page using **String.format**, this is on line 30.

We then use **restTemplate.getForObject** to make a get request to this URL and we specify the **User.class** as an input argument.

An object of this user class is what will hold the JSON response that we get from this URL. I've then added an artificial **Thread.sleep** here so that our async operations run slowly.

Once we've got the response for this user from GitHub, we call **CompletableFuture.completedFuture** and return this result.

5. Prepare Runner Class

Now, let's look at the **LookupAppRunner.java** file. This is the class that actually executes and invokes the LookupService to Lookup GitHub users.

```

1 package com.mytutorial.springbootasync.controller;
2
3 import java.util.concurrent.CompletableFuture;
4
5 import org.slf4j.Logger;
6 import org.slf4j.LoggerFactory;
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.boot.CommandLineRunner;
9 import org.springframework.stereotype.Component;
10
11 import com.mytutorial.springbootasync.model.User;
12 import com.mytutorial.springbootasync.service.LookupService;
13
14 @Component
15 public class LookupAppRunner implements CommandLineRunner {
16
17     private static final Logger logger = LoggerFactory.getLogger(LookupAppRunner.class);
18
19     @Autowired
20     private LookupService lookupService;
21
22     @Override
23     public void run(String... args) throws Exception {
24
25         CompletableFuture<User> info1 = lookupService.findUser("Pytorch");
26         CompletableFuture<User> info2 = lookupService.findUser("Tensorflow");
27         CompletableFuture<User> info3 = lookupService.findUser("Scikit-learn");
28         CompletableFuture<User> info4 = lookupService.findUser("Evanphx");
29         CompletableFuture<User> info5 = lookupService.findUser("Takeo");
30         CompletableFuture<User> info6 = lookupService.findUser("Macournoyer");
31
32         CompletableFuture.allOf(info1, info2, info3, info4, info5, info6).join();
33
34         logger.info("-->" + info1.get());
35         logger.info("-->" + info2.get());
36         logger.info("-->" + info3.get());
37         logger.info("-->" + info4.get());
38         logger.info("-->" + info5.get());
39         logger.info("-->" + info6.get());
40
41     }
42
43 }

```

The first thing you should notice about the LookupAppRunner is that it's tagged using the **@Component** annotation.

Indicating that it's a Spring managed object. It's Spring's job to take care of instantiating this object and also executing this object. But how do we know that this is an object that can be executed? Well, that's because the LookupAppRunner implements the CommandLineRunner interface.

A bean which implements this interface is basically an indication to Spring that once this bean has been instantiated, it should be a run or executed. As soon as your Spring Boot application starts up, the LookupAppRunner will be instantiated and then run. The method that will be invoked to run this application is the run method that you see here at the bottom.

In order to actually meet the asynchronous method calls, this LookupAppRunner needs the LookupService.

Which is automatically injected in thanks to the @Autowired annotation on the lookupService member variable.

Now, let's take a look at this run method which needs to be implemented as a part of the CommandLineRunner interface. This run takes in the command line arguments in the form of an array. We don't really need the command line arguments here.

What we do need is the lookupService. We'll look up six different users using lookupService.findUser. Pytorch, TensorFlow, Scikit-learn, spring-boot, spring-mvc and spring-security.

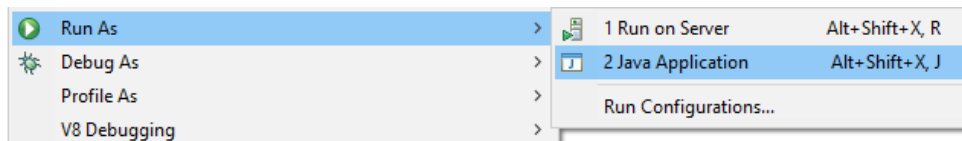
The findUser methods will all be run asynchronously. On line 34, the code CompletableFuture.allOf.join will ensure that all of the lookup operations are complete before we get to this point in the code.

Once all lookup operations are complete, we'll print out the six user's information that we have retrieved.

6. Run the App

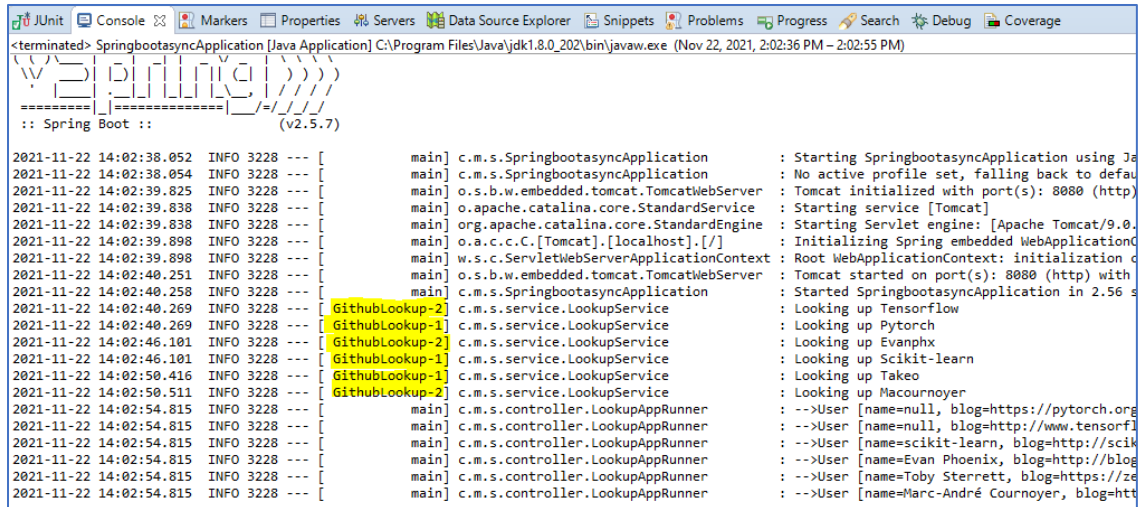
We're now ready to run the code for this Spring Boot application and see how asynchronous methods work.

Right click on SpringbootasyncApplication.java, select "Run As > Java Application"



The first thing you'll notice within the console logs is that we have logs for GitHubLookup-1 and 2. If you remember we had configured our thread pool executor to use exactly two threads.

Spring Boot Microservices: Asynchronous Methods, Schedulers, & Forms



```
<terminated> SpringbootasyncApplication [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Nov 22, 2021, 2:02:55 PM)
:: Spring Boot :: (v2.5.7)

2021-11-22 14:02:38.052 INFO 3228 --- [main] c.m.s.SpringbootasyncApplication : Starting SpringbootasyncApplication using Ja
2021-11-22 14:02:38.054 INFO 3228 --- [main] c.m.s.SpringbootasyncApplication : No active profile set, falling back to defau
2021-11-22 14:02:39.825 INFO 3228 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http
2021-11-22 14:02:39.838 INFO 3228 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-11-22 14:02:39.838 INFO 3228 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.
2021-11-22 14:02:39.898 INFO 3228 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationC
2021-11-22 14:02:40.251 INFO 3228 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization c
2021-11-22 14:02:40.258 INFO 3228 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with
2021-11-22 14:02:40.258 INFO 3228 --- [main] c.m.s.SpringbootasyncApplication : Started SpringbootasyncApplication in 2.56 s
2021-11-22 14:02:40.269 INFO 3228 --- [GithubLookup-2] c.m.s.service.LookupService : Looking up Tensorflow
2021-11-22 14:02:40.269 INFO 3228 --- [GithubLookup-1] c.m.s.service.LookupService : Looking up Pytorch
2021-11-22 14:02:46.101 INFO 3228 --- [GithubLookup-2] c.m.s.service.LookupService : Looking up Evanphx
2021-11-22 14:02:46.101 INFO 3228 --- [GithubLookup-1] c.m.s.service.LookupService : Looking up Scikit-learn
2021-11-22 14:02:50.416 INFO 3228 --- [GithubLookup-1] c.m.s.service.LookupService : Looking up Takeo
2021-11-22 14:02:50.511 INFO 3228 --- [GithubLookup-2] c.m.s.service.LookupService : Looking up Macournoyer
2021-11-22 14:02:54.815 INFO 3228 --- [main] c.m.s.controller.LookupAppRunner : -->User [name=null, blog=https://pytorch.org
2021-11-22 14:02:54.815 INFO 3228 --- [main] c.m.s.controller.LookupAppRunner : -->User [name=null, blog=http://www.tensorf
2021-11-22 14:02:54.815 INFO 3228 --- [main] c.m.s.controller.LookupAppRunner : -->User [name=scikit-learn, blog=http://scik
2021-11-22 14:02:54.815 INFO 3228 --- [main] c.m.s.controller.LookupAppRunner : -->User [name=Evan Phoenix, blog=http://blog
2021-11-22 14:02:54.815 INFO 3228 --- [main] c.m.s.controller.LookupAppRunner : -->User [name=Toby Sterrett, blog=https://ze
2021-11-22 14:02:54.815 INFO 3228 --- [main] c.m.s.controller.LookupAppRunner : -->User [name=Marc-André Cournoyer, blog=htt
```

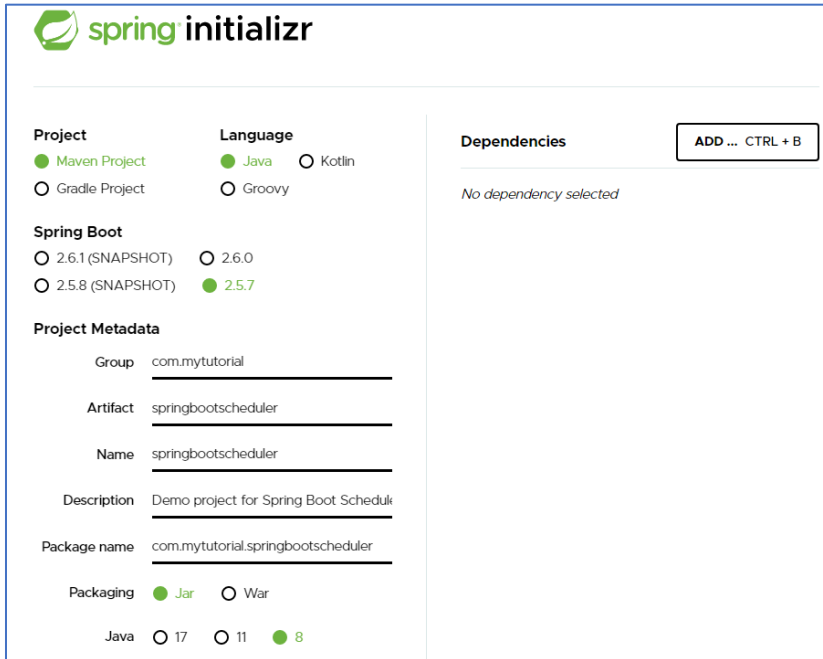
The first thread is called **GitHubLookup-1**. The second one is called **GitHubLookup-2**. Additional threads will not be spawned so we only use two threads to perform all of the lookups.

Once the threads performing the asynchronous find user lookups are complete. We'll see console log messages for the six user's information that we've retrieved from GitHub. Notice that for every user, we have the name of the user, the blog, the type of the user, and the URL.

Schedule Task

Now it's possible that you want your Spring Boot application to run certain tasks at scheduled intervals. It's possible to do this using Spring Boot using the **@Scheduled** annotation and that's exactly what we'll look at here in this demo.

1. Create Spring Boot Project



The image shows the Spring Initializr web application interface for creating a new project. The form is divided into several sections:

- Project:** ☒ Maven Project, ☐ Gradle Project
- Language:** ☒ Java, ☐ Kotlin, ☐ Groovy
- Spring Boot:** ☐ 2.6.1 (SNAPSHOT), ☐ 2.6.0, ☒ 2.5.8 (SNAPSHOT), ☒ 2.5.7
- Project Metadata:**
 - Group:
 - Artifact:
 - Name:
 - Description:
 - Package name:
- Packaging:** ☒ Jar, ☐ War
- Java:** ☐ 17, ☐ 11, ☒ 8
- Dependencies:** A section with an "ADD ... CTRL + B" button and the text "No dependency selected".

Within the **pom.xml** file, the only dependency descriptor I specify here is the spring-boot-starter.

pom.xml

```
springbootscheduler/pom.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.5.7</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.mytutorial</groupId>
12  <artifactId>springbootscheduler</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>springbootscheduler</name>
15  <description>Demo project for Spring Boot Scheduler</description>
16  <properties>
17    <java.version>1.8</java.version>
18  </properties>
19  <dependencies>
20    <dependency>
21      <groupId>org.springframework.boot</groupId>
22      <artifactId>spring-boot-starter</artifactId>
23    </dependency>
24
25    <dependency>
26      <groupId>org.springframework.boot</groupId>
27      <artifactId>spring-boot-starter-test</artifactId>
28      <scope>test</scope>
29    </dependency>
30  </dependencies>
31
32  <build>
33    <plugins>
34      <plugin>
35        <groupId>org.springframework.boot</groupId>
36        <artifactId>spring-boot-maven-plugin</artifactId>
37      </plugin>
38    </plugins>
39  </build>
40
41 </project>
```

Our application that will run tasks at a scheduled interval will be a simple Spring Boot application. We won't be using the web dependency, Thymeleaf or any other dependency.

2. Enabling Scheduler in Entry Point for Spring Boot Application

SpringbootschedulerApplication.java file as the main entry point of our app.

```
SpringbootschedulerApplication.java
1 package com.mytutorial.springbootscheduler;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.scheduling.annotation.EnableScheduling;
6
7 @SpringBootApplication
8 @EnableScheduling
9 public class SpringbootschedulerApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(SpringbootschedulerApplication.class, args);
13     }
14
15 }
```

There are a couple of interesting details to note here within this file. The first is that our scheduler application annotated using **@SpringBootApplication** has an additional annotation, **@EnableScheduling**.

This **@EnableScheduling** annotation is what tells Spring that this particular application will run tasks at scheduled intervals. This annotation will bring up a task executor to run the scheduled task on a background thread, not on the main thread. Another interesting detail to note here is that when we call **SpringApplication.run** on line 12, we simply run the application.

Make sure you don't close the application after you complete running.

3. Configure the Interval timing with type FixedRate

In order for this application to run scheduled tasks, the application has to be up and running at all points in time. The scheduled task will run at fixed intervals or at fixed delays based on your specification. Setting up a scheduled task is very straightforward and only requires the use of the **@Scheduled** annotation.

SimpleScheduler.java

```
SimpleScheduler.java
1 package com.mytutorial.springbootscheduler;
2
3 import java.text.SimpleDateFormat;
4 import java.util.Date;
5
6 import org.springframework.scheduling.annotation.Scheduled;
7 import org.springframework.stereotype.Component;
8
9 @Component
10 public class SimpleScheduler {
11
12     private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");
13
14     @Scheduled(fixedRate = 3000)
15     public void scheduleLookup() {
16         System.out.println(String.format("The time is now : %s", dateFormat.format(new Date())));
17     }
18
19 }
```

This has been tagged using the **@Component** annotation indicating that this is a component managed by the Spring runtime.

Within this component, we have the `ScheduledLookup` method which is tagged using the **@Scheduled** annotation. Anything with the **@Scheduled** annotation, Spring will invoke this method at pre-specified times based on a schedule.

Now there are different input arguments that you can specify to this **@Scheduled** annotation. To indicate how often this method will be invoked and when.

```
@Scheduled(fixedRate = 3000)
public void scheduleLookup() {
```

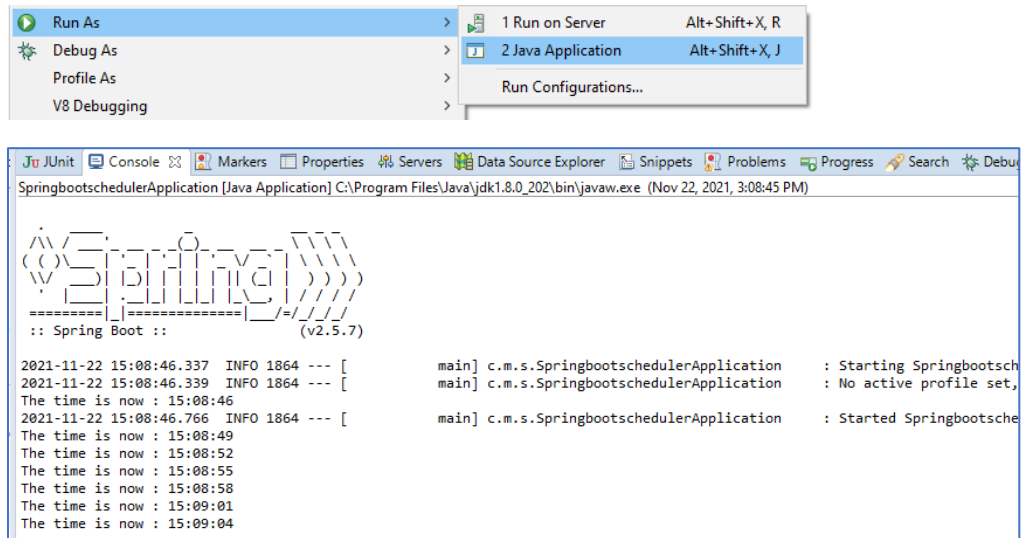
We have specified an argument **fixedRate**, which is equal to *3000*. This means that this method will be invoked every *3 seconds*, every *3000 milliseconds*. This **fixedRate** is typically used when every invocation of this method is an independent process. Subsequent invocations don't depend on previous invocations. There's exactly one background thread running the schedule tasks, which means that unless the previous invocation of the method is complete. The next invocation will not be started, no matter what value you specified for **fixedRate**.

The actual code within this method doesn't really do that much, we simply print out the current time in the hour minute second format.

4. Run and test the Application

Let's now run this code and see the output of our fixedRate scheduler.

Right Click on Entry Point of Application Class and select “Run As > Java Application”



If you take a look at the console log output, you'll see that every three seconds, the time is printed out to screen. The code for the method executes in no time at all. It doesn't take very much processing. The first method was invoked at second 46, then 49, then 52, then 55, then 58, and so on and so forth.

5. Update the interval execution type with fixed Delay

I'll now update the code in our **SimpleScheduler** to change when exactly this method will be executed. Now instead of **fixedRate**, I've specified a value for **fixedDelay**.

SimpleScheduler.java

```
SimpleScheduler.java
1 package com.mytutorial.springboot.scheduler;
2
3 import java.text.SimpleDateFormat;
4 import java.util.Date;
5
6 import org.springframework.scheduling.annotation.Scheduled;
7 import org.springframework.stereotype.Component;
8
9 @Component
10 public class SimpleScheduler {
11
12     private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");
13
14     @Scheduled(fixedDelay = 3000)
15     public void scheduleLookup() throws InterruptedException {
16         System.out.println(String.format("The time is now : %s", dateFormat.format(new Date())));
17
18         Thread.sleep(5000);
19     }
20 }
21 }
```

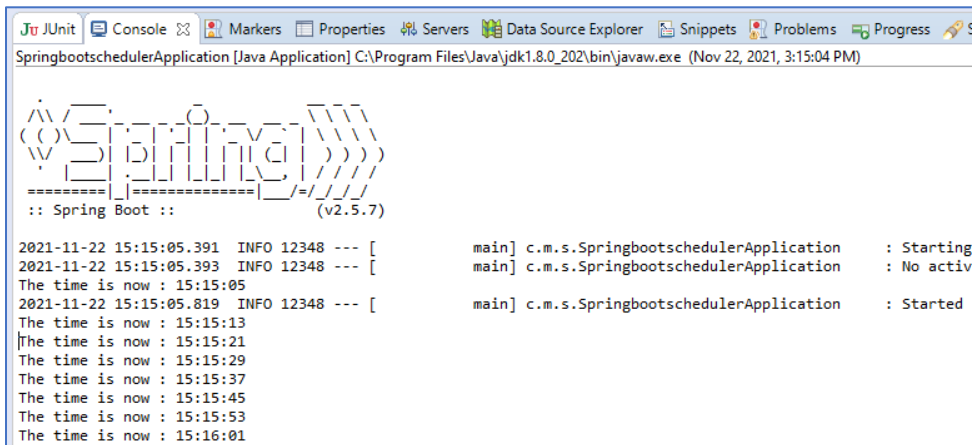
fixedDelay is equal to 3000.

```
@Scheduled(fixedDelay = 3000)
public void scheduleLookup() throws InterruptedException {
```

The fixedDelay works a little differently from the fixedRate that we saw earlier. This allows us to configure the delay between the completion of a previous execution of this method and start of the next execution. The method itself can take an arbitrary amount of time to execute. But the next execution of the method will not start unless three seconds have elapsed. The actual code within this method is still the same with a small but important difference. We'll print out the current time, the hh mm ss format. I'll then call Thread.sleep for five seconds so that this method takes at least five seconds to execute.

The introduction of the sleep here will allow us to see exactly how fixedDelay works.

Run this code and let's take a look at the times at which the method was executed.



```
SpringbootSchedulerApplication [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Nov 22, 2021, 3:15:04 PM)

:: Spring Boot :: (v2.5.7)

2021-11-22 15:15:05.391 INFO 12348 --- [main] c.m.s.SpringbootSchedulerApplication : Starting
2021-11-22 15:15:05.393 INFO 12348 --- [main] c.m.s.SpringbootSchedulerApplication : No activ
The time is now : 15:15:05
2021-11-22 15:15:05.819 INFO 12348 --- [main] c.m.s.SpringbootSchedulerApplication : Started
The time is now : 15:15:13
The time is now : 15:15:21
The time is now : 15:15:29
The time is now : 15:15:37
The time is now : 15:15:45
The time is now : 15:15:53
The time is now : 15:16:01
```

The first execution was at the 05 second, the method itself would have taken five seconds to run because of our Thread.sleep. The next execution was five plus three, eight seconds after the first

execution at 13. The second execution would have taken five seconds to complete as well. The next execution after that is at 21, that is eight seconds after the second execution. You can see that thus `fixedDelay` introduces a delay of three seconds between the completion of the previous invocation of this method. And the start of the next invocation.

6. Update the interval execution type with fixed Rate and initial Delay

Let's update the configuration of our **@Scheduled** annotation once again. This time I've specified a fixedRate of 3000 milliseconds and an initial delay of 5000 milliseconds.

SimpleScheduler.java

```
SimpleScheduler.java
1 package com.mytutorial.springboot.scheduler;
2
3 import java.text.SimpleDateFormat;
4 import java.util.Date;
5
6 import org.springframework.scheduling.annotation.Scheduled;
7 import org.springframework.stereotype.Component;
8
9 @Component
10 public class SimpleScheduler {
11
12     private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");
13
14     @Scheduled(fixedRate = 3000, initialDelay = 5000)
15     public void scheduleLookup() throws InterruptedException {
16         System.out.println(String.format("The time is now : %s", dateFormat.format(new Date())));
17
18         Thread.sleep(5000);
19     }
20 }
21 }
```

Within the code, I've also caused the current thread to sleep for five seconds. Now there are two things to watch out for here.

```
@Scheduled(fixedRate = 3000, initialDelay = 5000)
public void scheduleLookup() throws InterruptedException {
```

fixedRate basically means that this method will be invoked every three seconds but this method will take at least five seconds to complete. How will fixedRate work then? That's the first detail to watch out for.

The second one is that the invocation of this method won't start up till at least five seconds after the app is up and running. That is what the initial delay is for. It'll be hard to see how initialDelay works in this demo. But we'll see how fixedRate works when the time taken for the execution of the method is higher than the rate that we have specified.

```
SpringbootSchedulerApplication [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Nov 22, 2021, 3:21:39 PM)

:: Spring Boot :: (v2.5.7)

2021-11-22 15:21:40.884 INFO 26600 --- [main] c.m.s.SpringbootSchedulerApplication : Starting Spring
2021-11-22 15:21:40.886 INFO 26600 --- [main] c.m.s.SpringbootSchedulerApplication : No active profile
2021-11-22 15:21:41.299 INFO 26600 --- [main] c.m.s.SpringbootSchedulerApplication : Started Spring
The time is now : 15:21:46
The time is now : 15:21:51
The time is now : 15:21:56
The time is now : 15:22:01
The time is now : 15:22:06
The time is now : 15:22:11
The time is now : 15:22:16
```

You can see that the last log statement was at 21 minutes and 41 seconds.

Then there was a delay of 5 seconds and the first invocation of the method was at 21 minutes and 46 seconds. Even though we specified a `fixedRate` of three seconds, the method takes five seconds to execute. So the next invocation of the method is at 21 minutes and 51 seconds, then 21 minutes and 56 seconds and so on. There is exactly one background thread running this method, so if the method takes longer than the rate that we have specified. The previous invocation of the method has to be completed before the method is invoked again.

7. Update the interval execution type with Fixed Rate String

Let's now make one last change to our `@Scheduled` annotation.

```
SimpleScheduler.java
1 package com.mytutorial.springboot.scheduler;
2
3 import java.text.SimpleDateFormat;
4 import java.util.Date;
5
6 import org.springframework.scheduling.annotation.Scheduled;
7 import org.springframework.stereotype.Component;
8
9 @Component
10 public class SimpleScheduler {
11
12     private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");
13
14     @Scheduled(fixedRateString = "${scheduler.rate}")
15     public void scheduleLookup() throws InterruptedException {
16         System.out.println(String.format("The time is now : %s", dateFormat.format(new Date())));
17
18         Thread.sleep(5000);
19     }
20 }
21 }
```

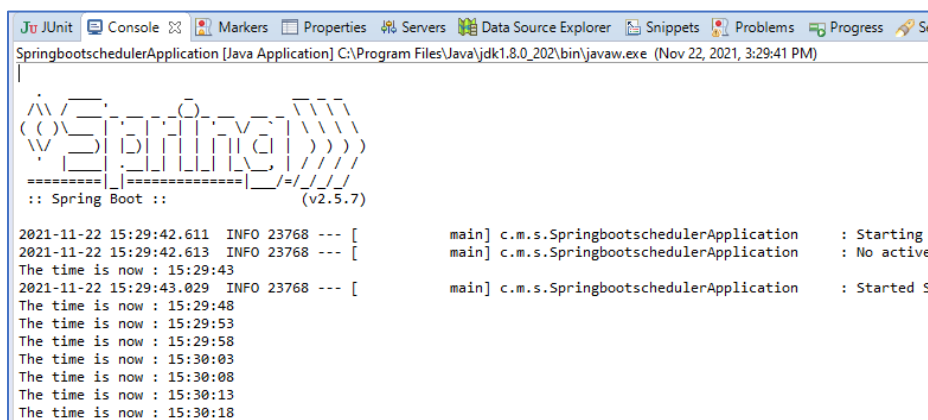
I've specified the `fixedRate` in a string format. This is extremely useful if you want to specify the rate for your tasks using application properties. Here, the `scheduler.rate` property is what we determine how often and at what rate our method will be invoked.

```
@Scheduled(fixedRateString = "${scheduler.rate}")
public void scheduleLookup() throws InterruptedException {
```

Let's head over to **application.properties** and I've specified the `sheduler.rate` as 4000 milliseconds.

```
SimpleScheduler.java application.properties
1
2 scheduler.rate=4000
```

Let's run this code and our method is invoked as we would expect, every four seconds.



```
SpringbootSchedulerApplication [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Nov 22, 2021, 3:29:41 PM)

:: Spring Boot :: (v2.5.7)

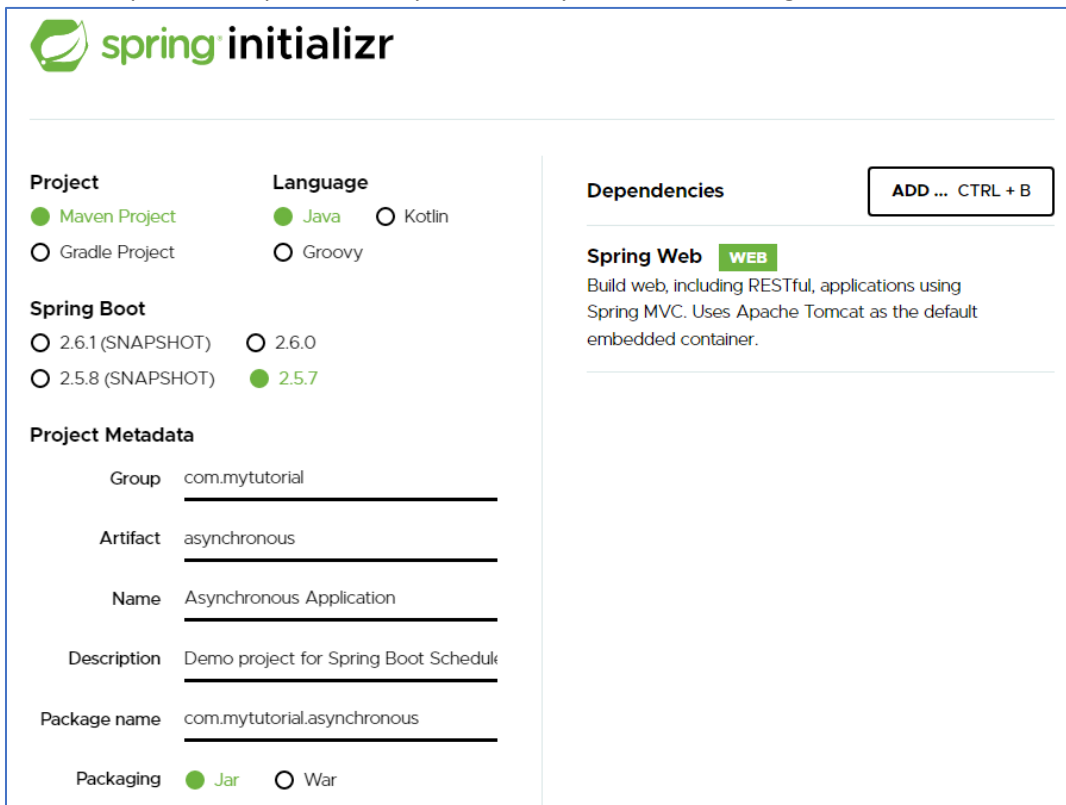
2021-11-22 15:29:42.611 INFO 23768 --- [main] c.m.s.SpringbootSchedulerApplication : Starting
2021-11-22 15:29:42.613 INFO 23768 --- [main] c.m.s.SpringbootSchedulerApplication : No active
The time is now : 15:29:43
2021-11-22 15:29:43.029 INFO 23768 --- [main] c.m.s.SpringbootSchedulerApplication : Started S
The time is now : 15:29:48
The time is now : 15:29:53
The time is now : 15:29:58
The time is now : 15:30:03
The time is now : 15:30:08
The time is now : 15:30:13
The time is now : 15:30:18
```

Scheduling Asynchronous Tasks

In this demo, we'll bring together some of the concepts that we've studied earlier in this learning path. We'll see how we can run scheduled tasks in Spring Boot, but we'll have these scheduled tasks run on different background threads asynchronously.

1. Create Spring Boot Project

We're going to use the rest template to make an asynchronous request to GitHub URLs. We'll have multiple web requests run asynchronously on different background threads.



The image shows the Spring Initializr web application interface for creating a new project. The interface is divided into several sections:

- Project:** Includes radio buttons for **Maven Project** (selected) and **Gradle Project**.
- Language:** Includes radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for versions **2.6.1 (SNAPSHOT)**, **2.6.0**, **2.5.8 (SNAPSHOT)**, and **2.5.7** (selected).
- Project Metadata:** A form with the following fields:
 - Group:** `com.mytutorial`
 - Artifact:** `asynchronous`
 - Name:** `Asynchronous Application`
 - Description:** `Demo project for Spring Boot Schedule`
 - Package name:** `com.mytutorial.asynchronous`
 - Packaging:** Includes radio buttons for **Jar** (selected) and **War**.
- Dependencies:** A section with a button **ADD ... CTRL + B** and a list of dependencies. The first dependency is **Spring Web**, which is highlighted with a green **WEB** tag. Below it, a description reads: "Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container."

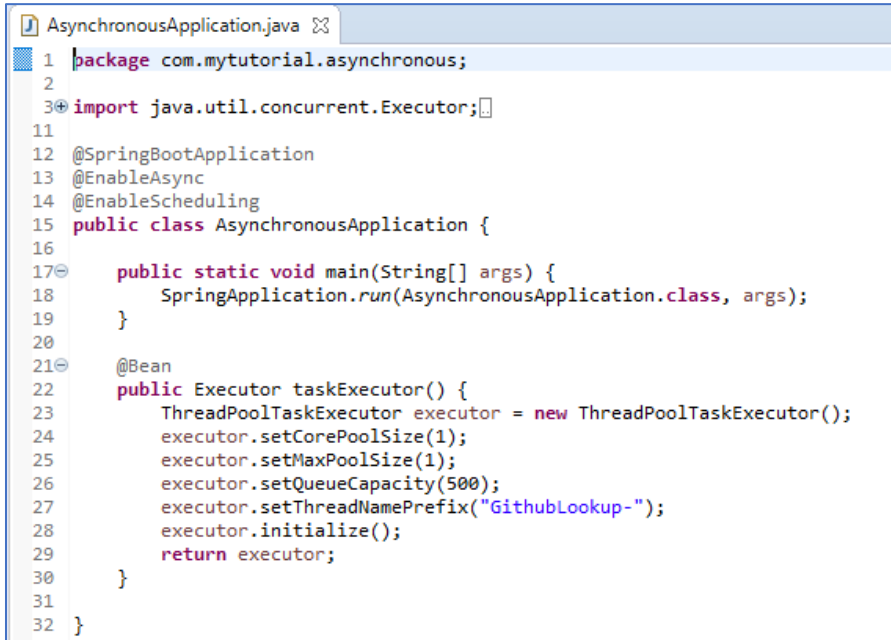
Here is the **pom.xml** file for this demo.

```
asynchronous/pom.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4     <modelVersion>4.0.0</modelVersion>
5     <parent>
6         <groupId>org.springframework.boot</groupId>
7         <artifactId>spring-boot-starter-parent</artifactId>
8         <version>2.5.7</version>
9         <relativePath/> <!-- lookup parent from repository -->
10    </parent>
11    <groupId>com.mytutorial</groupId>
12    <artifactId>asynchronous</artifactId>
13    <version>0.0.1-SNAPSHOT</version>
14    <name>Asynchronous Application</name>
15    <description>Demo project for Spring Boot Scheduled Asynchronous</description>
16    <properties>
17        <java.version>1.8</java.version>
18    </properties>
19    <dependencies>
20        <dependency>
21            <groupId>org.springframework.boot</groupId>
22            <artifactId>spring-boot-starter-web</artifactId>
23        </dependency>
24
25        <dependency>
26            <groupId>org.springframework.boot</groupId>
27            <artifactId>spring-boot-starter-test</artifactId>
28            <scope>test</scope>
29        </dependency>
30    </dependencies>
31
32    <build>
33        <plugins>
34            <plugin>
35                <groupId>org.springframework.boot</groupId>
36                <artifactId>spring-boot-maven-plugin</artifactId>
37            </plugin>
38        </plugins>
39    </build>
40
41 </project>
```

You can see that we depend on the **spring-boot-starter-web** dependency descriptor.

2. Enable Synchronous and Scheduler in Entry Point Spring Boot Application

We'll now look at the main entry point of our application, the **AsynchronousApplication.java**.



```
1 package com.mytutorial.asynchronous;
2
3 import java.util.concurrent.Executor;
4
11
12 @SpringBootApplication
13 @EnableAsync
14 @EnableScheduling
15 public class AsynchronousApplication {
16
17     public static void main(String[] args) {
18         SpringApplication.run(AsynchronousApplication.class, args);
19     }
20
21     @Bean
22     public Executor taskExecutor() {
23         ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
24         executor.setCorePoolSize(1);
25         executor.setMaxPoolSize(1);
26         executor.setQueueCapacity(500);
27         executor.setThreadNamePrefix("GithubLookup-");
28         executor.initialize();
29         return executor;
30     }
31
32 }
```

You can see that the **AsynchronousApplication** has several annotations, but each of these annotations is familiar to us. We have the **@SpringBootApplication** annotation indicating this is the entry point. We have **@EnableAsync**, allowing us to run methods asynchronously on background threads within this app. And finally, this class also has the **@EnableScheduling** annotation, allowing us to run scheduled tasks at specified time intervals or time delays.

Thus, by using all of these different annotations, Spring Boot allows us to wire up different features within the same application. And this makes Spring Boot very powerful indeed, the main method of this application simply calls **SpringApplication.run**.

Let's scroll down a bit, and here we've specified a bean for the **taskExecutor**.

If you remember, we had mentioned earlier that the **@EnableAsync** annotation, specifically looks for a method with this name, **taskExecutor** on line 22. This is a bean, which means it will be managed by the Spring runtime. The **TaskExecutor** returns an object of the **executor** interface which allows us to use a thread pool executor to run our tasks on different threads.

I've instantiated a new thread pool executor, but I've set the *CorePoolSize* and the *MaxPoolSize* to be exactly one to start off with.

We've configured this thread pool to have exactly one thread, which means our method invocations will run on just one thread.

3. Prepare Model Object

When we make HTTP GET request to the GitHub user's URL, we get a JSON response. Spring can automatically convert the JSON response to a Java object. And every GitHub user will be represented as an object of this User class.

User.java

```
User.java
1 package com.mytutorial.asynchronous.model;
2
3 import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
4
5 @JsonIgnoreProperties(ignoreUnknown = true)
6 public class User {
7
8     private String name;
9     private String blog;
10    private String type;
11    private String url;
12    public String getName() {
13        return name;
14    }
15    public void setName(String name) {
16        this.name = name;
17    }
18    public String getBlog() {
19        return blog;
20    }
21    public void setBlog(String blog) {
22        this.blog = blog;
23    }
24    public String getType() {
25        return type;
26    }
27    public void setType(String type) {
28        this.type = type;
29    }
30    public String getUrl() {
31        return url;
32    }
33    public void setUrl(String url) {
34        this.url = url;
35    }
36
37    @Override
38    public String toString() {
39        return String.format("User [name=%s, blog=%s, type=%s, url=%s ]",
40            name, blog, type, url);
41    }
42
43 }
```

We've applied the **@JsonIgnoreProperties ignoreUnknown=true** annotation indicating that this object should ignore those JSON properties that haven't been explicitly defined as member variables.

The only properties we're interested in are the name, blog, type and url for all of the GitHub users. Make sure that you have the getters and setters for each of these properties.

And at the bottom, we have overridden the toString implementation of the object base class.

4. Prepare Business Logic

The actual business logic to look up a particular GitHub user is within this LookupService class.

LookupService.java

```

1 package com.mytutorial.asynchronous.service;
2
3 import java.util.concurrent.CompletableFuture;
4
5 import org.slf4j.Logger;
6 import org.slf4j.LoggerFactory;
7 import org.springframework.boot.web.client.RestTemplateBuilder;
8 import org.springframework.scheduling.annotation.Async;
9 import org.springframework.stereotype.Service;
10 import org.springframework.web.client.RestTemplate;
11
12 import com.mytutorial.asynchronous.model.User;
13
14 @Service
15 public class LookupService {
16
17     private static final Logger logger = LoggerFactory.getLogger(LookupService.class);
18
19     private static final String GITHUB_USERS_URL = "https://api.github.com/users/%s";
20
21     private final RestTemplate restTemplate;
22
23     public LookupService(RestTemplateBuilder restTemplateBuilder) {
24         this.restTemplate = restTemplateBuilder.build();
25     }
26
27     @Async
28     public CompletableFuture<User> findUser(String user) throws InterruptedException {
29         logger.info(String.format("Looking up %s", user));
30         String url = String.format(GITHUB_USERS_URL, user);
31         User result = restTemplate.getForObject(url, User.class);
32         Thread.sleep(1000L);
33         return CompletableFuture.completedFuture(result);
34     }
35
36 }

```

This class has been annotated using the **@Service** annotation, indicating intent indicating that this is a Spring bean containing business logic.

Here is our GITHUB_USERS_URL = "<https://api.github.com/users/%s>" which will allow us to retrieve the information for individual users.

We have the RestTemplate, that is injected as an input argument to the constructor of this LookupService. This RestTemplate is what allows us to make HTTP GET request to specified URLs. If you scroll further down, we have the findUser method, which, as you can see from the @Async annotation on it, will be run on a background thread.

The name of the user is an input argument. We generate the GitHub URL using **String.format**, then we use **restTemplate.getForObject**, specify the URL. The JSON response from the URL is then converted to an object of the user class.

We'll then have this thread sleep for a second to simulate a little bit of delay.

And once we've got the user response, we'll return a CompletableFuture with this response.

5. Prepare Controller

In order to invoke this find user method, we'll set up a scheduled task and we'll do this within the LookupController. The LookupController I have specified as a component, which Spring manages, it has a **@Component** annotation.

```

1 package com.mytutorial.asynchronous;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.concurrent.CompletableFuture;
6
7 import org.slf4j.Logger;
8 import org.slf4j.LoggerFactory;
9 import org.springframework.beans.factory.annotation.Autowired;
10 import org.springframework.scheduling.annotation.Scheduled;
11 import org.springframework.stereotype.Component;
12
13 import com.mytutorial.asynchronous.model.User;
14 import com.mytutorial.asynchronous.service.LookupService;
15
16 @Component
17 public class LookupController {
18
19     private static final Logger logger = LoggerFactory.getLogger(LookupController.class);
20
21     @Autowired
22     private LookupService lookupService;
23
24     private static int userIndex = 0;
25
26     private static final List<String> userList = new ArrayList<>();
27
28     static {
29         userList.add("Pytorch");
30         userList.add("Tensorflow");
31         userList.add("Scikit-learn");
32         userList.add("Evanphx");
33         userList.add("Takeo");
34         userList.add("Macournoyer");
35     }
36
37     @Scheduled(fixedRate = 2000)
38     public void scheduledTasks() throws Exception {
39         CompletableFuture<User> info = lookupService.findUser(userList.get(userIndex));
40         userIndex = (userIndex + 1) % userList.size();
41         logger.info("---> " + info.get());
42     }
43
44 }

```

This LookupController contains a reference to the LookupService, and I have used the **@Autowired** annotation so that Spring injects this automatically. I also have a userIndex initialized to 0 to start off with and the list of users that we'll look up on GitHub.

We'll initialize the users list within a static initialization block. We call userList.add, we'll add Pytorch, TensorFlow, Scikit-learn, and so on. All of these are open source projects available on GitHub.

And we'll look up this information. Notice that my scheduled tasks method has an **@Scheduled** annotation with a **fixedRate** set to 2000 milliseconds or two seconds.

This method will be invoked every two seconds, so long as our application is running. We then use the LookupService to find the user at the specified index. We use userList.get(userIndex) for the specified user.

Thanks to the **@Async** annotation on the find user method, findUser will run on a background thread, which means that the scheduled task will complete very quickly doesn't have much processing to do.

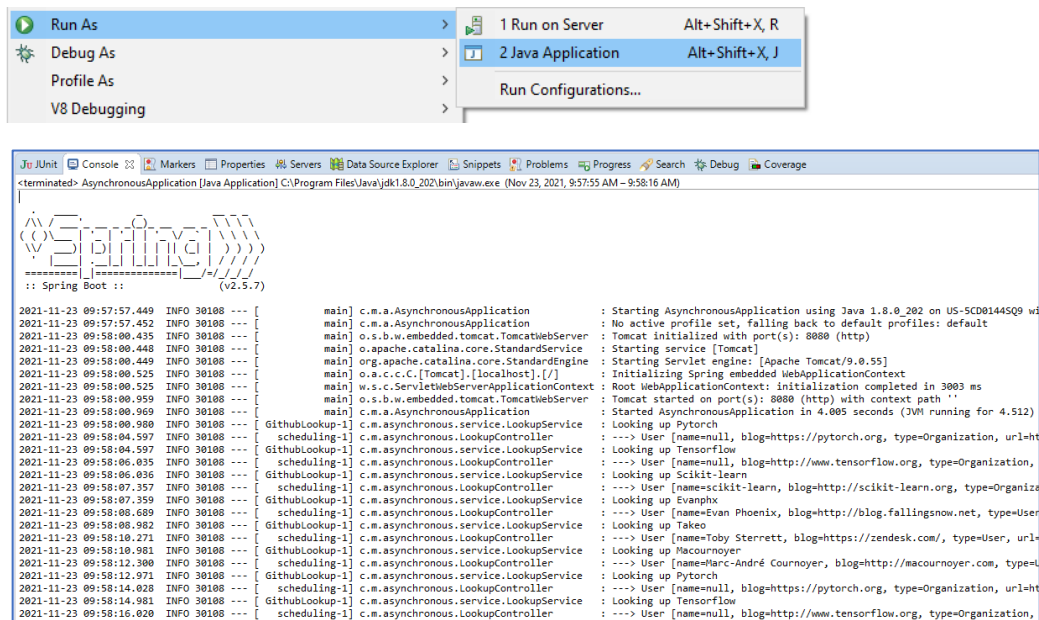
Once we find the user at a particular index, I'll increment user index by 1, modulo the usersList.size. So once we've reach the end of our users, we'll wrap back again to the beginning. We'll then print out onto the screen using logger.info the current user.

And this is how we run scheduled tasks asynchronously.

6. Run And Test Spring Boot Asynchronous Scheduled Application

Go ahead and run this code and let's take a look at the output here within our console window.

Right Click on Entry Point of Application Class and select “Run As > Java Application”



The screenshot shows an IDE with the 'Run As' context menu open, highlighting '2 Java Application'. Below it, the console window displays the output of the application. The output shows the application starting successfully on port 8080. It then enters a loop where it looks up users from GitHub and logs the results. The logs show that only one thread is running, which is the 'main' thread, and it is performing all the lookups and scheduling tasks.

```

2021-11-23 09:57:57.449 INFO 30108 --- [main] c.m.a.AsynchronousApplication : Starting AsynchronousApplication using Java 1.8.0_202 on US-5CD81445Q9 w...
2021-11-23 09:57:57.452 INFO 30108 --- [main] c.m.a.AsynchronousApplication : No active profile set, falling back to default profiles: default
2021-11-23 09:58:00.435 INFO 30108 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2021-11-23 09:58:00.448 INFO 30108 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-11-23 09:58:00.449 INFO 30108 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.55]
2021-11-23 09:58:00.525 INFO 30108 --- [main] o.s.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-11-23 09:58:00.525 INFO 30108 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 3003 ms
2021-11-23 09:58:00.959 INFO 30108 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2021-11-23 09:58:00.969 INFO 30108 --- [main] c.m.a.AsynchronousApplication : Started AsynchronousApplication in 4.005 seconds (JVM running for 4.512)
2021-11-23 09:58:00.980 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up Pytorch
2021-11-23 09:58:04.597 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up Tensorflow
2021-11-23 09:58:06.035 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up Scikit-Learn
2021-11-23 09:58:06.036 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up User [name=scikit-learn, blog=http://scikit-learn.org, type=Organization, url=https://scikit-learn.org]
2021-11-23 09:58:07.359 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up Evanphx
2021-11-23 09:58:08.689 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up User [name=Evan Phoenix, blog=http://blog.fallingsnow.net, type=User, url=https://blog.fallingsnow.net]
2021-11-23 09:58:08.982 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up Takeo
2021-11-23 09:58:10.271 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up User [name=Toby Sterrett, blog=https://zendesk.com/, type=User, url=https://zendesk.com/]
2021-11-23 09:58:10.981 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up MacCounoyer
2021-11-23 09:58:12.300 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up User [name=Marc-André Cournoyer, blog=http://maccounoyer.com, type=User, url=http://maccounoyer.com]
2021-11-23 09:58:12.971 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up Pytorch
2021-11-23 09:58:14.981 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up Tensorflow
2021-11-23 09:58:16.020 INFO 30108 --- [c.m.a.s.s.LookupService] c.m.a.s.s.LookupService : Looking up User [name=null, blog=http://www.tensorflow.org, type=Organization, url=https://www.tensorflow.org]
  
```

The first thing that you'll observe here is that we have exactly one thread running. Notice all of the logs GitHub lookup and scheduling is all for just thread one. That's because we've configured our thread pool executor to have just one thread. Each time we look up a user, we get the response, we log the response out to screen.

7. Update And Increase thread Schedule

I'm now going to kill the running of this application and head over to the **AsynchronousApplication.java** file, and configure my TaskExecutor to have two threads within the thread pool. So change the CorePoolSize and the MaxPoolSize, set them both to 2.

```

1 package com.mytutorial.asynchronous;
2
3 import java.util.concurrent.Executor;
4
5 import org.springframework.boot.SpringApplication;
6 import org.springframework.boot.autoconfigure.SpringBootApplication;
7 import org.springframework.context.annotation.Bean;
8 import org.springframework.scheduling.annotation.EnableAsync;
9 import org.springframework.scheduling.annotation.EnableScheduling;
10 import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
11
12 @SpringBootApplication
13 @EnableAsync
14 @EnableScheduling
15 public class AsynchronousApplication {
16
17     public static void main(String[] args) {
18         SpringApplication.run(AsynchronousApplication.class, args);
19     }
20
21     @Bean
22     public Executor taskExecutor() {
23         ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
24         executor.setCorePoolSize(2);
25         executor.setMaxPoolSize(2);
26         executor.setQueueCapacity(500);
27         executor.setThreadNamePrefix("GithubLookup-");
28         executor.initialize();
29         return executor;
30     }
31
32 }

```

Now our lookup operations will be performed on two threads asynchronously at a scheduled time. Go ahead and run this code and immediately you'll see within the console log messages are lookups.

```

<terminated> AsyncronousApplication [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\java.exe (Nov 23, 2021, 10:02:53 AM - 10:02:54 AM)

<img alt="Spring Boot logo" data-bbox="111 115 250 210"/>

:: Spring Boot :: (v2.5.7)

2021-11-23 10:02:53.643 INFO 31024 --- [main] c.m.a.AsyncronousApplication : Starting AsyncronousApplication using Java 1.8.0_202 on US-5CD0144509 w
2021-11-23 10:02:53.647 INFO 31024 --- [main] c.m.a.AsyncronousApplication : No active profile set, falling back to default profiles: default
2021-11-23 10:02:53.724 INFO 31024 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2021-11-23 10:02:53.734 INFO 31024 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-11-23 10:02:53.735 INFO 31024 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.55]
2021-11-23 10:02:53.810 INFO 31024 --- [main] w.s.c.c.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-11-23 10:02:53.810 INFO 31024 --- [main] w.s.c.c.[Tomcat].[localhost].[/] : Root WebApplicationContext: initialization completed in 310 ms
2021-11-23 10:02:53.258 INFO 31024 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2021-11-23 10:02:53.270 INFO 31024 --- [main] c.m.a.AsyncronousApplication : Started AsyncronousApplication in 4.157 seconds (JVM running for 4.683)
2021-11-23 10:02:53.282 INFO 31024 --- [GithubLookup-1] c.m.asyncronous.service.LookupService : c.m.asyncronous.service.LookupService
2021-11-23 10:02:42.885 INFO 31024 --- [scheduling-1] c.m.asyncronous.LookupController : --- User [name=null, blog=https://pytorch.org, type=Organization, url=htt
2021-11-23 10:02:44.887 INFO 31024 --- [GithubLookup-2] c.m.asyncronous.service.LookupService : Looking up TensorFlow
2021-11-23 10:02:44.194 INFO 31024 --- [scheduling-1] c.m.asyncronous.LookupController : --- User [name=null, blog=http://www.tensorflow.org, type=Organization, u
2021-11-23 10:02:44.195 INFO 31024 --- [GithubLookup-1] c.m.asyncronous.service.LookupService : Looking up Scikit-learn
2021-11-23 10:02:45.500 INFO 31024 --- [scheduling-1] c.m.asyncronous.LookupController : --- User [name=scikit-learn, blog=http://scikit-learn.org, type=Organizat
2021-11-23 10:02:45.502 INFO 31024 --- [GithubLookup-2] c.m.asyncronous.service.LookupService : Looking up Evamphx
2021-11-23 10:02:46.085 INFO 31024 --- [scheduling-1] c.m.asyncronous.LookupController : --- User [name=Evan Phoenix, blog=https://blog.fallingsnow.net, type=User,
2021-11-23 10:02:47.269 INFO 31024 --- [GithubLookup-1] c.m.asyncronous.service.LookupService : Looking up Takeo
2021-11-23 10:02:48.584 INFO 31024 --- [scheduling-1] c.m.asyncronous.LookupController : --- User [name=Toby Sterrett, blog=https://zendesk.com/, type=User, url=
2021-11-23 10:02:49.269 INFO 31024 --- [GithubLookup-2] c.m.asyncronous.service.LookupService : Looking up Macyorch
2021-11-23 10:02:50.571 INFO 31024 --- [scheduling-1] c.m.asyncronous.LookupController : --- User [name=Marc-André Cournoyer, blog=http://macournoyer.com, type=US
2021-11-23 10:02:51.277 INFO 31024 --- [GithubLookup-2] c.m.asyncronous.service.LookupService : Looking up Pytorch
2021-11-23 10:02:52.321 INFO 31024 --- [scheduling-1] c.m.asyncronous.LookupController : --- User [name=https://pytorch.org, type=Organization, url=htt
2021-11-23 10:02:53.277 INFO 31024 --- [GithubLookup-2] c.m.asyncronous.service.LookupService : Looking up TensorFlow
2021-11-23 10:02:54.328 INFO 31024 --- [scheduling-1] c.m.asyncronous.LookupController : --- User [name=null, blog=http://www.tensorflow.org, type=Organization,
```

Use two threads, thread one and thread two. Having many more threads running in the background will basically allow us to retrieve and process the URLs faster.

Using Request Parameters and Dynamic Paths

In this demo, we'll see how we can work with values associated with the request parameters in a URL.

1. Create New Spring Boot Project

I start off on the Spring Initializr page to set up a new project. **start.spring.io** is the URL for this page. I want a Maven Project, I want to code in Java, and I want to use Spring Boot version 2.3.1. I'll now scroll down below in order to specify the other metadata associated with my project.

The group is *com.mytutorial*, the artifact and name will be *springboot*. Spring Initializr then generates a package name for me, *com.mytutorial.springboot*.

The packaging is Jar and the Java version that I'm using is Java 8. There are a few starter dependency descriptors that I want in my pom.xml. Click on ADD DEPENDENCIES, you can either search for

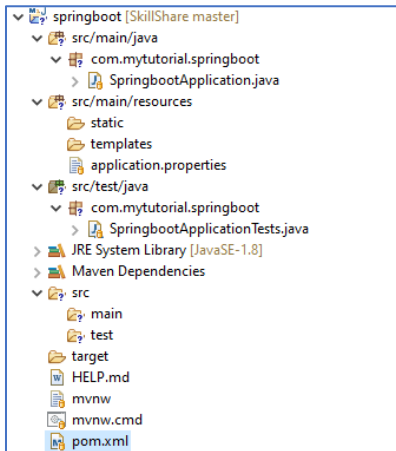
a specific dependency or you can scroll down and find it. Here is Spring Web, I click on it, select it and add it as a dependency for my project. There is one more dependency I'll add in before we move on, click on ADD DEPENDENCIES. And you can either search for the Thymeleaf template engine. Or you can scroll down and you'll find it under TEMPLATE ENGINES. Select Thymeleaf, add that to your Spring Boot project as well.

The screenshot shows the Spring Initializr web form. The form is divided into several sections:

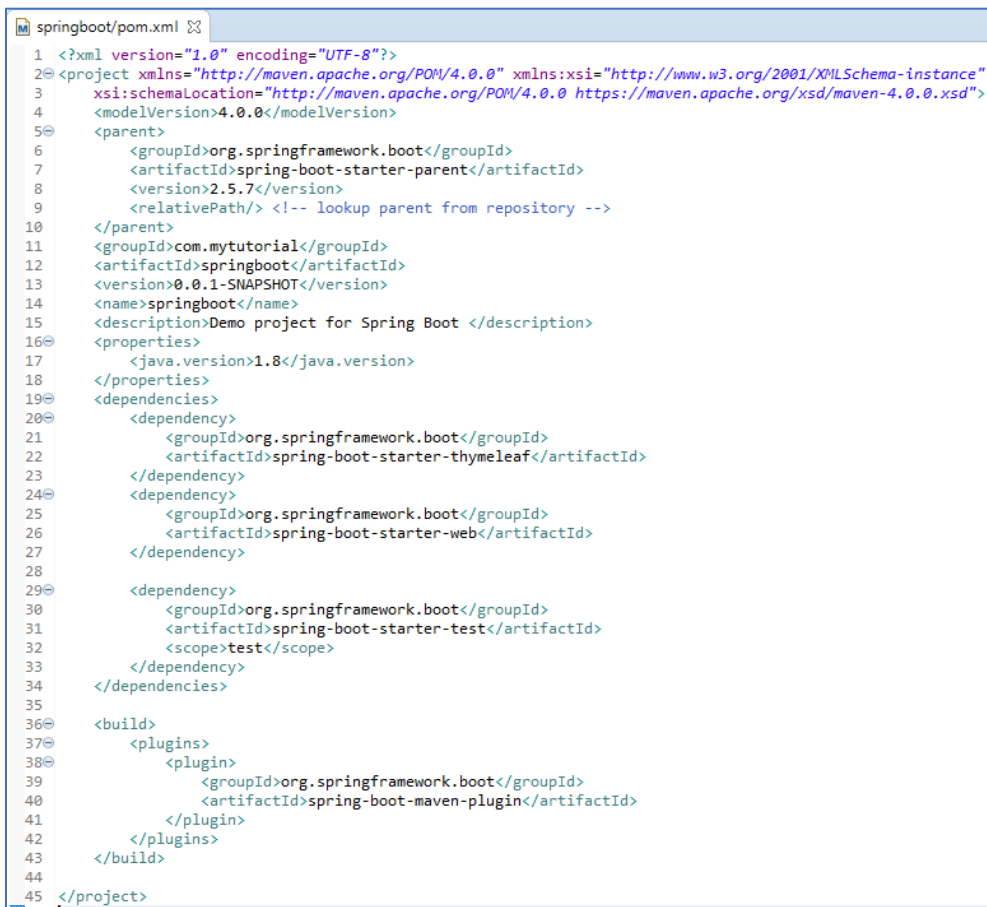
- Project:** ☒ Maven Project, ☐ Gradle Project
- Language:** ☒ Java, ☐ Kotlin, ☐ Groovy
- Spring Boot:** ☐ 2.6.1 (SNAPSHOT), ☐ 2.6.0, ☐ 2.5.8 (SNAPSHOT), ☒ 2.5.7
- Project Metadata:**
 - Group:
 - Artifact:
 - Name:
 - Description:
 - Package name:
 - Packaging: ☒ Jar, ☐ War
- Dependencies:** A button labeled "ADD ... CTRL + B" is present. Below it, two dependencies are listed:
 - 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.

At the bottom of the form, there are three buttons: "GENERATE CTRL + G", "EXPLORE CTRL + SPACE", and "SHARE..."

Now, all that you need to do is click on the **GENERATE** button. And the project template with all of its structure will be downloaded as a Zip file onto your local machine. I'll then unzip the Zip file and import it as an Eclipse project.



Let's take a look at the structure of the project that we have just generated. And then we can look at code. I'm going to select pom.xml and within this form, you will find the **spring-boot-starter-parent** artifact on line 7.



We always inherit from this parent for Spring Boot. And the two dependencies that we specified are also here, **spring-boot-starter-thymeleaf** and **spring-boot-starter-web**.

2. Entry Point Spring Boot Application

Let's now take a look at the **SpringbootApplication.java** file, which is the main entry point of our application. It's decorated using the annotation **@SpringBootApplication**. And we call `SpringApplication.run` on this class.

```
SpringbootApplication.java
1 package com.mytutorial.springboot;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class SpringbootApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(SpringbootApplication.class, args);
11     }
12
13 }
```

3. Prepare Controller Object

Within the package, *com.mytutorial.springboot.controller*, I have defined a class that represents a rest controller. Notice that this web service controller has the **@RestController** annotation.

WebServiceController.java

```
WebServiceController.java
1 package com.mytutorial.springboot.controller;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RequestParam;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 @RequestMapping(value = "/organization/")
10 public class WebServiceController {
11
12     @GetMapping("/info")
13     public String myName(@RequestParam(defaultValue = "Skillsoft") String name) {
14         return "<h2> Welcome to Spring Boot at <i>" + name + "</i>! This is an Http Get Request</h2>";
15     }
16 }
```

When you tag your controller class as a **RestController**. Any value that you return from your handler methods is automatically considered a response body. That is, it's considered to be a web response that is rendered to the user.

The return values from handler methods are not logical views. Rather, they are response bodies to be directly rendered rather than mapped to a physical view.

This controller is where we specify the methods used to handle the incoming web request. Notice that we have an **@RequestMapping** annotation on the web service controller class itself.

A `RequestMapping` annotation associated with the class is typically used. If all of the URLs for individual methods in this controller have the same prefix.

Here, all of the URLs will have the organization prefix. So this prefix path is specified as a part of the `RequestMapping` annotation associated with the class.

Within this class, I have a single method handler. The name of the method is `myName` and it's tagged using the `GetMapping /info`. So the path to invoke this particular method will be `/organization/info`.

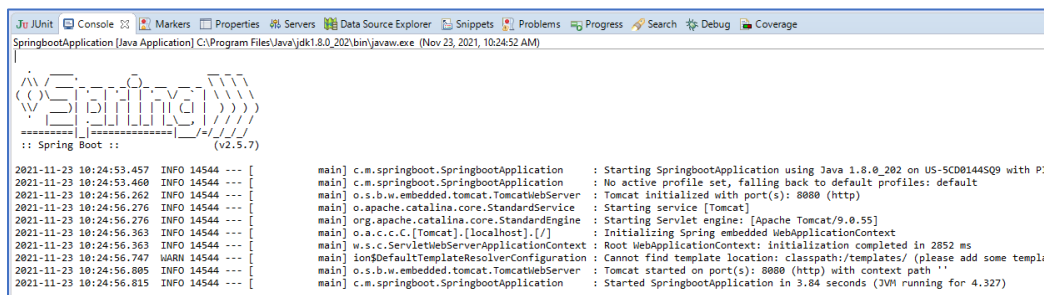
The request path that corresponds to this particular method gets its prefix from the **@RequestMapping** annotation. That we've applied on the class, that is `/organization`, plus the mapping that we have specified for the method itself, which is just `/info`.

Let's now turn our attention to the input arguments that we have specified to this particular method. It takes in a single input argument, the name. This is tagged using the **@RequestParam** annotation indicating that its value should be extracted from the request parameters to this request. If there is no request parameter with *name* as the key, the default value for this input argument will be *Skillsoft*. Now the body of this method is very straightforward.

We simply return an h2 header which says, Welcome to Spring Boot, with the name in italics, and this is an HTTP GET request.

And this is all there is to our application. Let's run this code and head over to the browser and hit `localhost.8080/organization/info`.

Right Click on “WebServiceController.java” and select “Run As > Java Application”

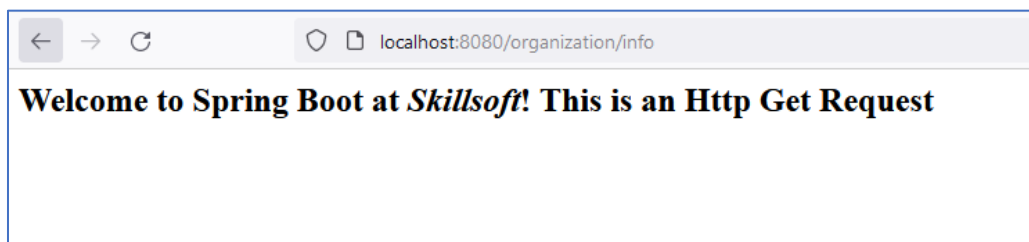


```

SpringBootApplication [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (Nov 23, 2021, 10:24:52 AM)

:: Spring Boot :: (v2.5.7)

2021-11-23 10:24:53.457 INFO 14544 --- [main] c.m.springboot.SpringBootApplication : Starting SpringBootApplication using Java 1.8.0_202 on US-5CD81445Q9 with P
2021-11-23 10:24:53.460 INFO 14544 --- [main] c.m.springboot.SpringBootApplication : No active profile set, falling back to default profiles: default
2021-11-23 10:24:56.262 INFO 14544 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2021-11-23 10:24:56.276 INFO 14544 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-11-23 10:24:56.276 INFO 14544 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.55]
2021-11-23 10:24:56.363 INFO 14544 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-11-23 10:24:56.363 INFO 14544 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 2852 ms
2021-11-23 10:24:56.747 WARN 14544 --- [main] ion$DefaultTemplateResolverConfiguration : Cannot find template location: classpath:/templates/ (please add some templ
2021-11-23 10:24:56.805 INFO 14544 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2021-11-23 10:24:56.815 INFO 14544 --- [main] c.m.springboot.SpringBootApplication : Started SpringBootApplication in 3.84 seconds (JVM running for 4.327)
  
```



There are a number of details to notice about this path here. Notice the complete path has the prefix `/organization`, and then `/info`. So this is a combination of the request mapping on the web service controller class and on the `MyName` method. Another thing to notice is that we've specified no value for the request parameter name, which means it'll take on the default value, *Skillsoft*. And that's what the result says, welcome to Spring Boot at Skillsoft.

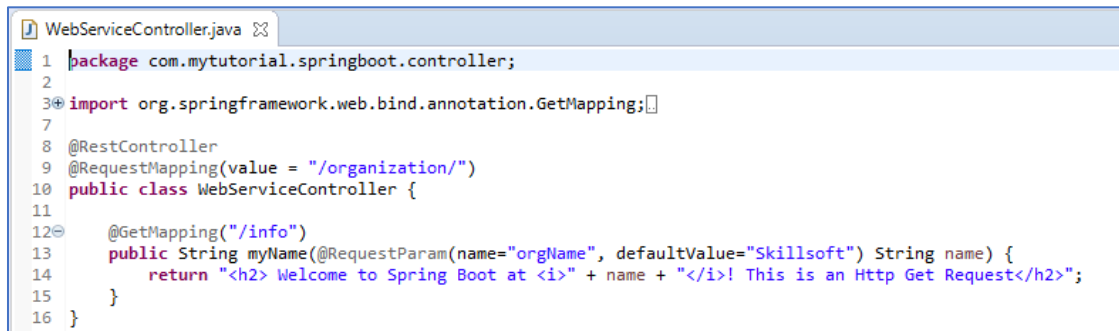
Let's try this once again. I'm going to head over to `localhost:8080/organization/info`. My request parameters come after the question mark `name=Loonycorn`.



We now have a value for the request parameter. Notice that the request parameter name is the same as the name of our variable. This is what Spring expects unless you specify otherwise. Now we'll get the result, Welcome to Spring Boot at Loonycorn!

4. Accepting Request Parameter name as Variable Input

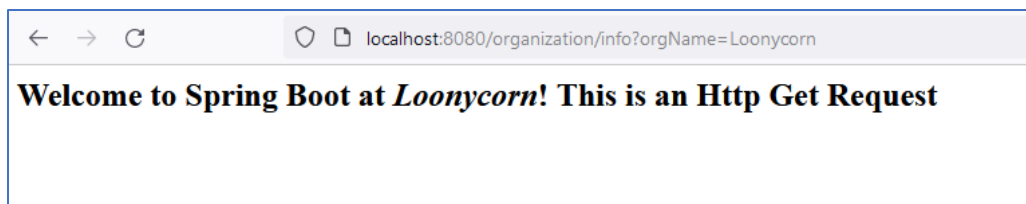
Back to Eclipse, and I've updated my code a little bit. What if I want my request parameter to be different from the name of the variable that I've used in my code? Let's see how we can handle that. The only change that I have made here in this code is in the **@RequestParam** annotation for the String name input argument.



```
1 package com.mytutorial.springboot.controller;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4
5
6
7
8 @RestController
9 @RequestMapping(value = "/organization/")
10 public class WebServiceController {
11
12     @GetMapping("/info")
13     public String myName(@RequestParam(name="orgName", defaultValue="Skillsoft") String name) {
14         return "<h2> Welcome to Spring Boot at <i>" + name + "</i>! This is an Http Get Request</h2>";
15     }
16 }
```

Notice that I have name=orgName and defaultValue=Skillsoft. When we specify a name for the request parameter, this is the name that this parameter will look for within the URL. So the parameter will now look for orgName rather than the variable name, which is just name. Now if this is confusing when we run this code, things will become much clearer.

I've run this code, let's head over to <http://localhost:8080>. Then I have the path itself, /organization/info, then question mark, then my request parameters, orgName=Loonycorn.



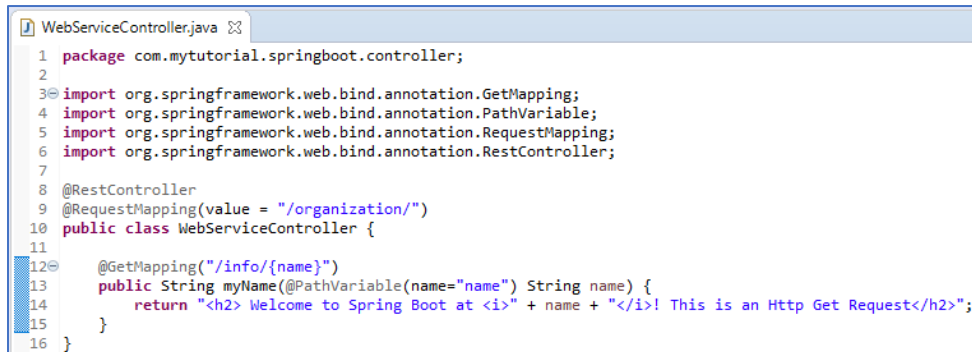
← → ↻ localhost:8080/organization/info?orgName=Loonycorn

Welcome to Spring Boot at *Loonycorn*! This is an Http Get Request

Notice that my RequestParam now called *orgName*, which is the name that we had specified in the **@RequestParam** annotation. The value associated with orgName which happens to be Loonycorn will be extracted and passed into the name variable. So we get Welcome to Spring Boot at Loonycorn.

5. Accepting Path Variable as Variable Input

Back to Eclipse where we'll change our code so that we can extract the name of the organization from a dynamic URL path. We'll do this using the **@PathVariable** annotation.



```

1 package com.mytutorial.springboot.controller;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.PathVariable;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 @RequestMapping(value = "/organization/")
10 public class WebServiceController {
11
12     @GetMapping("/info/{name}")
13     public String myName(@PathVariable(name="name") String name) {
14         return "<h2> Welcome to Spring Boot at <i>" + name + "</i>! This is an Http Get Request</h2>";
15     }
16 }

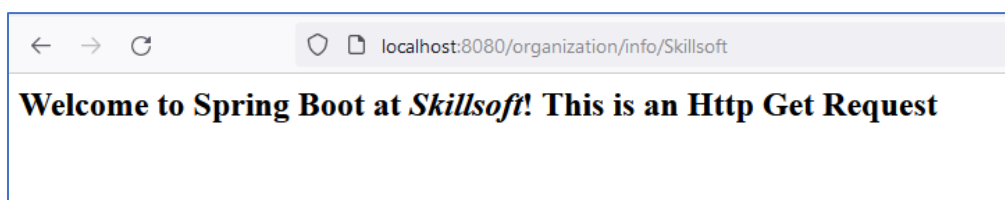
```

It's quite possible that within your web app, you want to structure your URL such that the information is a part of the path itself. Rather than a separate request parameter. Now in order to extract information from this dynamic path, we'll use the **@PathVariable** annotation. The first change that you should see here is in our **@GetMapping**. Notice that the GetMapping is now for /info/name within curly braces.

This indicates to Spring that the name portion of the path is actually dynamic. And can be any value, no matter what the value in the name portion of this path, map that particular path to this MyName handler method.

Because this dynamic portion will be extracted and input as an input argument to this method. Notice the string name here which is now annotated using **@PathVariable**. We configured this PathVariable to extract the name portion of the dynamic path and make it available in the name variable. The actual method itself is not that interesting, or rather it's the same as what we've seen before. Welcome to Spring Boot with the name that we've extracted from the dynamic path.

Now let's head over to <http://localhost:8080/organization/info/Skillsoft> after having run the server and let's hit localhost:8080/organization/info/Skillsoft. The dynamic portion of this URL is the name Skillsoft.



Thanks to the **@PathVariable** annotation, this dynamic portion associated with the name variable will be extracted. And passed as an input argument to our handler method. And this is what will give us the resulting page. Welcome to Spring Boot at Skillsoft, this is an HTTP GET request.

6. Accepting Form input

Now let's change our controller once again so that we specify an input using a form. Now there are several changes that we've made here to this controller method.

The first change that should jump out at you is that we've used the **@Controller** annotation for our web service controller.

WebServiceController.java

```

1 package com.mytutorial.springboot.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.GetMapping;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestParam;
7 import org.springframework.web.bind.annotation.ResponseBody;
8
9 @Controller
10 @RequestMapping(value = "/organization/")
11 public class WebServiceController {
12
13     @GetMapping("/")
14     public String formPage() {
15         return "editName";
16     }
17
18     @ResponseBody
19     @GetMapping("/info")
20     public String myName(@RequestParam(defaultValue = "Skillsoft") String name) {
21         return "<h2> Welcome to Spring Boot at <i>" + name + "</i>! This is an Http Get Request</h2>";
22     }
23 }

```

The use of the **@Controller** annotation rather than the **@RestController**. Tells Spring that the response from the handler methods should be treated as a *logical view by default*. Unless the handler method has been explicitly tagged using the **@ResponseBody** annotation.

If this seems confusing, don't worry. There's an example of it right here in front of you. Take a look at the first method here within this controller which has been annotated using `@GetMapping("/")`.

This is the method that renders a form page to the user. A page containing a form where you can specify an input, the return value is `"editName"`.

This is interpreted as a logical view name, which is mapped to a physical view name by the **Thymeleaf template engine**.

We have a second handler method here within this controller. This is a method that we've seen earlier, the **myName** method with the `@GetMapping("/info")`. It takes in a request parameter, the name with a default value Skillsoft.

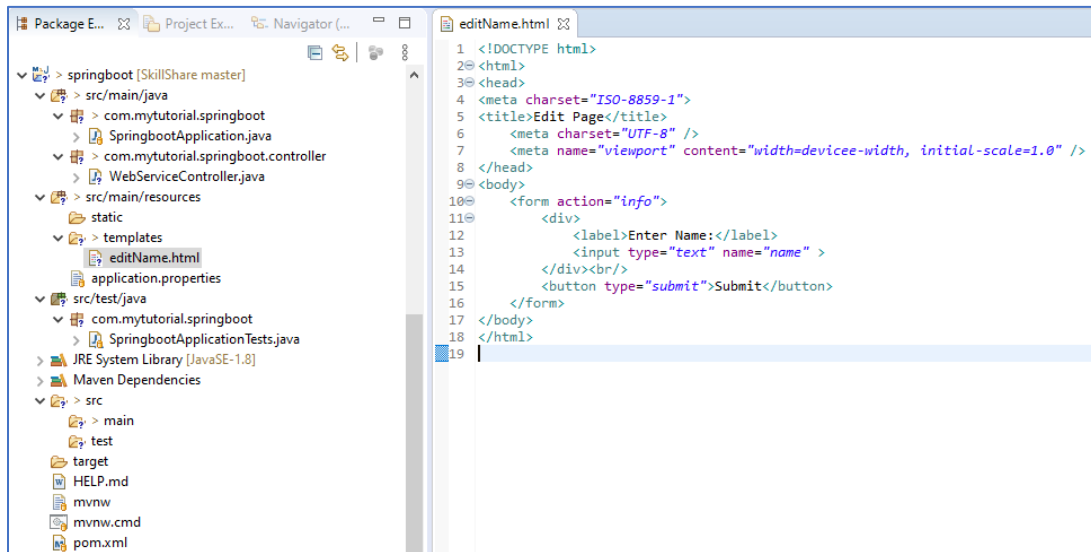
```
public String myName(@RequestParam(defaultValue = "Skillsoft") String name)
```

Notice that I have specified an additional annotation to this method, the **@ResponseBody** annotation.

This basically tells Spring that the response of this particular method in this controller should be rendered as a web response. It's not a logical view name but the response itself. When you use

the **@Controller** annotation on the class. You need to explicitly tag your methods using **@ResponseBody** if they return a web response rather than a logical view name.

Let's take a look at the physical view representation of the edit name logical view. Under the *templates* folder, I have an **editName.html** file. And this file contains a form which we can use to fill in the name that will be passed as a request parameter to our handler methods.



```

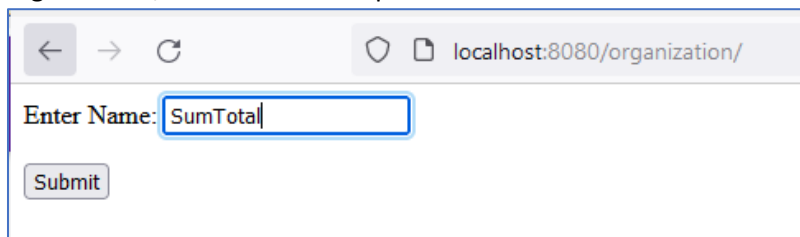
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="ISO-8859-1">
5 <title>Edit Page</title>
6 <meta charset="UTF-8" />
7 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
8 </head>
9 <body>
10 <form action="/info">
11 <div>
12 <label>Enter Name:</label>
13 <input type="text" name="name" />
14 </div><br/>
15 <button type="submit">Submit</button>
16 </form>
17 </body>
18 </html>
19

```

Notice that we have a form on line 12. The action is `info`, indicating that it'll hit the relative path `/info`.

It has a single input box on line 16 of type `text`, `name=name`. This name should match the request parameter that our back end expects. And we have a submit button.

Let's run this code and head over to `localhost:8080/organization`, our `editname.html` form will be displayed. Let's enter a name *SumTotal* here and hit Submit and this will take us to the page `organization/info` with name equal to *SumTotal*.




Using Form

In this demo, we'll see how we can work with forms in Spring Boot, not just simple forms, but complex forms with many fields and many different kinds of UI widgets.

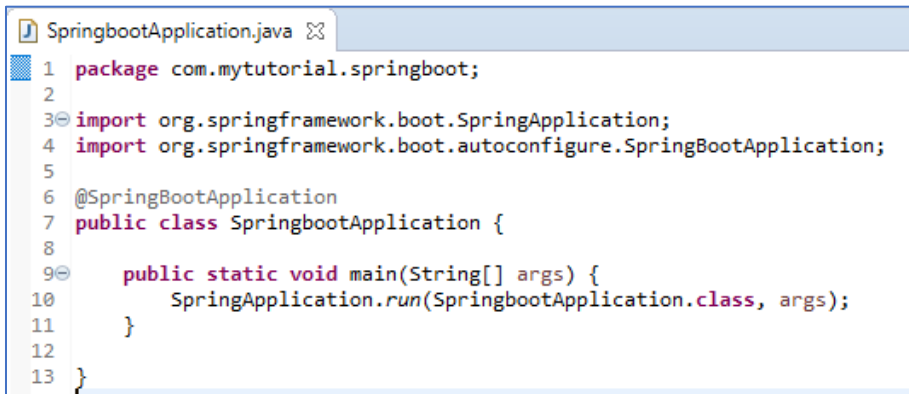
7. Inspect Maven Dependency in pom.xml

This is what our **pom.xml** looks like. Make sure that you include the **spring-boot-starter-thymeleaf** and the **spring-boot-starter-web** dependency descriptors.

```
springboot/pom.xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4    <modelVersion>4.0.0</modelVersion>
5    <parent>
6      <groupId>org.springframework.boot</groupId>
7      <artifactId>spring-boot-starter-parent</artifactId>
8      <version>2.5.7</version>
9      <relativePath/> <!-- lookup parent from repository -->
10   </parent>
11   <groupId>com.mytutorial</groupId>
12   <artifactId>springboot</artifactId>
13   <version>0.0.1-SNAPSHOT</version>
14   <name>springboot</name>
15   <description>Demo project for Spring Boot</description>
16   <properties>
17     <java.version>1.8</java.version>
18   </properties>
19   <dependencies>
20     <dependency>
21       <groupId>org.springframework.boot</groupId>
22       <artifactId>spring-boot-starter-thymeleaf</artifactId>
23     </dependency>
24     <dependency>
25       <groupId>org.springframework.boot</groupId>
26       <artifactId>spring-boot-starter-web</artifactId>
27     </dependency>
28   </dependencies>
29   <build>
30     <plugins>
31       <plugin>
32         <groupId>org.springframework.boot</groupId>
33         <artifactId>spring-boot-maven-plugin</artifactId>
34       </plugin>
35     </plugins>
36   </build>
37 </project>
```


8. Inspect Entry Point of Spring Boot Application

The main entry point of our application is not very different. We have the **SpringbootApplication** class annotated using **@SpringBootApplication**, which calls `SpringApplication.run`.

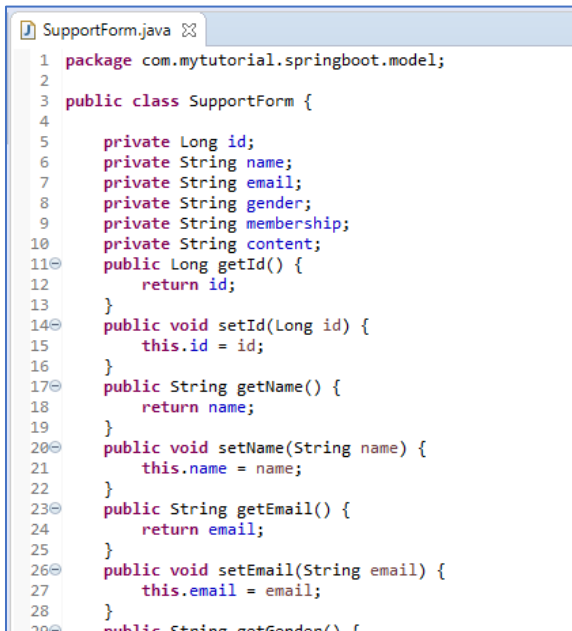


```
SpringbootApplication.java
1 package com.mytutorial.springboot;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class SpringbootApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(SpringbootApplication.class, args);
11     }
12
13 }
```

9. Create Form Model Object

When you work with forms, you're essentially accepting user data that you then send to the back end and maybe store in a database.

So working with forms involves a user interface, some business logic and data. This means we'll be making extensive use of this Spring MVC module which gives us the model view controller paradigm. Let's take a look at the model object, which is this **SupportForm.java**. We've defined this SupportForm within the model package. And it's a plain old Java object or a POJO, whose member variables correspond to all of the forms data that we want our user to fill in.



```
1 package com.mytutorial.springboot.model;
2
3 public class SupportForm {
4
5     private Long id;
6     private String name;
7     private String email;
8     private String gender;
9     private String membership;
10    private String content;
11    public Long getId() {
12        return id;
13    }
14    public void setId(Long id) {
15        this.id = id;
16    }
17    public String getName() {
18        return name;
19    }
20    public void setName(String name) {
21        this.name = name;
22    }
23    public String getEmail() {
24        return email;
25    }
26    public void setEmail(String email) {
27        this.email = email;
28    }
29    public String getGender() {
```

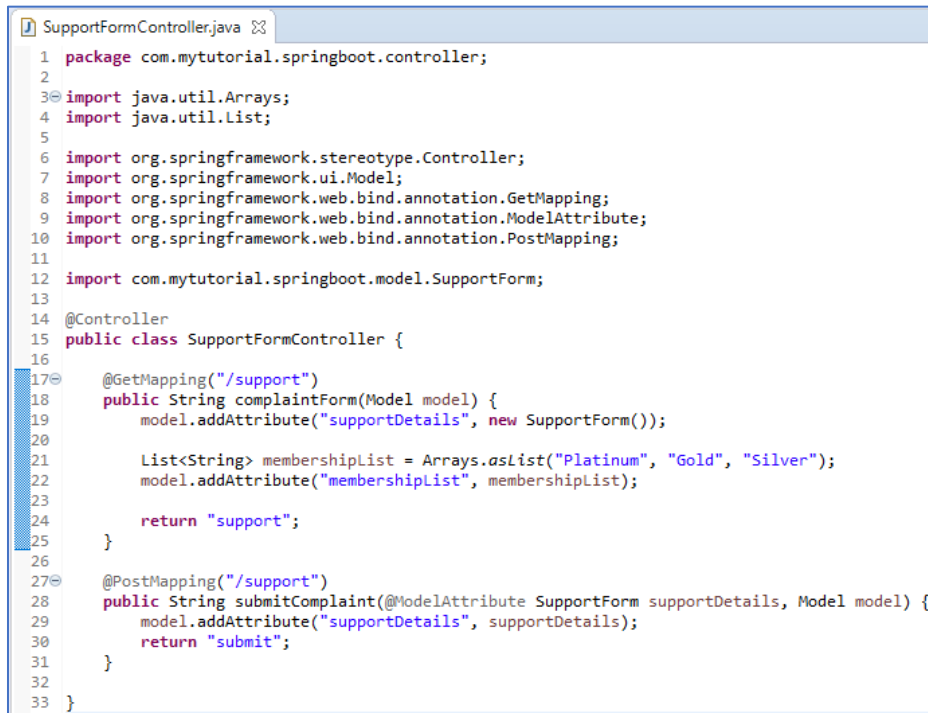
Let's take a look at the member variables here. We have an id, name, email, gender, membership, and content.

All of these are simple strings and all of this information the user will have to fill in using our form. So this plain old Java object is the model object backing the form that'll bind to our form.

When we work with forms in Spring, we instantiate a model object and then bind that model to the form. The inputs that we specify to the form fields will be used to set values on this model object. This SupportForm object itself is very simple. You can see that we have the member variables and then we have the getters and setters for all of these member variables. This is a POJO, there's nothing really special about this object.

10. Create Controller

Let's move on and take a look at the controller within our MVC application. The controller, as you know, specifies the methods that are invoked corresponding to incoming requests. We have a **SupportFormController** tagged using the **@Controller** annotation,



```

1 package com.mytutorial.springboot.controller;
2
3 import java.util.Arrays;
4 import java.util.List;
5
6 import org.springframework.stereotype.Controller;
7 import org.springframework.ui.Model;
8 import org.springframework.web.bind.annotation.GetMapping;
9 import org.springframework.web.bind.annotation.ModelAttribute;
10 import org.springframework.web.bind.annotation.PostMapping;
11
12 import com.mytutorial.springboot.model.SupportForm;
13
14 @Controller
15 public class SupportFormController {
16
17     @GetMapping("/support")
18     public String complaintForm(Model model) {
19         model.addAttribute("supportDetails", new SupportForm());
20
21         List<String> membershipList = Arrays.asList("Platinum", "Gold", "Silver");
22         model.addAttribute("membershipList", membershipList);
23
24         return "support";
25     }
26
27     @PostMapping("/support")
28     public String submitComplaint(@ModelAttribute SupportForm supportDetails, Model model) {
29         model.addAttribute("supportDetails", supportDetails);
30         return "submit";
31     }
32
33 }

```

which means the return values from the handler methods are logical views by default.

The basic idea of this demo is that the user will be presented with a support form, which he'll then fill in.

```

@GetMapping("/support")
public String complaintForm(Model model) {
    model.addAttribute("supportDetails", new SupportForm());
    List<String> membershipList = Arrays.asList(
        "Platinum", "Gold", "Silver");

    model.addAttribute("membershipList", membershipList);
    return "support";
}

```

The first method I have here **complaintForm** is tagged using the **@GetMapping** annotation for the path (**/support**). This is the get request that we make to actually render the **SupportForm()** to the user.

Let's take a look at the contents of this method. It takes us an input argument the **model** of the application. The views in our application will render the data represented in the model. Before we render the page that holds the form, we need to specify the object that will be bound to that

form. This is our **SupportForm** object. We instantiate a new **SupportForm** on line 19. And we use **model.addAttribute** to send the **SupportForm** down to the user interface in the **supportDetails** parameter. Before you render a form page in Spring Boot, you need to instantiate the object that will be bound to that form, our **SupportForm** object.

In addition, we'll send down some additional information for our view to render as well, the **membershipList**. This will be a drop down that will be used to set the membership field on our **SupportForm**. The possible values for this drop down are *Platinum*, *Gold* and *Silver*. We set that up in the form of a list and then use **model.addAttribute** to pass this **membershipList** down to our form view.

Once we have all of the model details that our form needs, we render the support logical view which maps to a physical view *support.html*.

```
@PostMapping("/support")
public String submitComplaint(@ModelAttribute SupportForm supportDetails,
    Model model) {
    model.addAttribute("supportDetails", supportDetails);
    return "submit";
}
```

Now the next method here is the **submitComplaint** method, which is mapped to the path */support* once again, but it responds to post request. You can see the **@PostMapping** on line 27.

So you can see on line 17 and line 27 that we can have two methods mapped to the same path, so long as they respond to different HTTP request. The one on line 17 responds to Get request and the one on line 27 responds to Post request.

Now, whenever the user clicks Submit on a form, a Post request is made to the back end. And this is what this *submitComplaint* method will handle. The input arguments to this method are the *SupportForm* and the *Model* object.

Observe that the **SupportForm** that is injected into this method has been tagged using the **@ModelAttribute** annotation. This essentially means that this is the bound object that we get from our form on the client sent back to the server. The fields of this **SupportForm** object will be filled in based on the values that the user has specified on the client form. All of the user's details will be available in the **SupportForm** object. I then add these details using the **Model** and then render the **Submit** page which will simply print out the user's details.

11. Create View Page

Now it's time for us to look at the UI code. The **support.html** file contains the user form, which a user will fill in regarding his customer support issue.

support.html

```

1 <!DOCTYPE html>
2 <html xmlns:th="https://www.thyaleaf.org">
3 <head>
4 <meta charset="ISO-8859-1">
5 <title>Nile E-Commerce Customer Support Form</title>
6 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7 </head>
8 <style>
9     form div {
10         display: table-row;
11         padding: 4px;
12     }
13     label, input, select {
14         display: table-cell;
15         margin: 5px;
16         text-align: left;
17     }
18     form div div {
19         display: table-cell;
20     }
21 </style>
22 <body>
23 <h2>Nile E-Commerce Customer Support Form</h2>
24 <form action="#" th:action="@{/support}" th:object="${supportDetails}" method="post">
25     <div>
26         <label>Customer Id:</label>
27         <input type="text" th:field="*{id}" />
28     </div>
29     <div>
30         <label>Name:</label>
31         <input type="text" th:field="*{name}" />
32     </div>
33     <div>
34         <label>Email:</label>
35         <input type="text" th:field="*{email}" />
36     </div>
37     <div>
38         <label>Gender:</label>
39         <input type="radio" th:field="*{gender}" value="Male" />Male
40         <input type="radio" th:field="*{gender}" value="Female" />Female
41     </div>
42     <div>
43         <label>Membership:</label>
44         <select th:field="*{membership}">
45             <option th:each="ListItem : ${membershipList}" th:value="${ListItem}" th:text="${ListItem}" />
46         </select>
47     </div>
48     <div>
49         <label>Request:</label>
50         <input type="text" th:field="*{content}" />
51     </div>
52     <p><input type="submit" value="submit" /><input type="reset" value="Reset" /></p>
53 </form>
54 </body>
55 </html>

```

We have some style CSS at the very top here. And here is our Nile E-Commerce Customer Support Form.

```

<form action="#" th:action="@{/support}"
th:object="${supportDetails}" method="post">

```

We have the form tag on line 24. We've set the **action** attribute to **"#"**. We don't want the default form action to do anything. We want to configure a **thymeleaf action** for this form. You can see the **th:action** has been set to **/support**. This is the path to which a post request will be made when we submit this form.

Next we've specified `th:object="${supportDetails}"`. This is the object that our form has been bound to, this **SupportForm** object that we instantiated in Controller Class and sent down via the Model to the client.

SupportFormController.java

```
14 @Controller
15 public class SupportFormController {
16
17     @GetMapping("/support")
18     public String complaintForm(Model model) {
19         model.addAttribute("supportDetails", new SupportForm());
20
21         List<String> membershipList = Arrays.asList("Platinum", "Gold", "Silver");
22         model.addAttribute("membershipList", membershipList);
23
24         return "support";
25     }
26 }
```

The new **SupportForm** object that we have instantiated in the `complaintForm` method of the server is sent down to the client and bound to this form using `th:object`.

And this form will be submitted using a post method, `method="post"`.

Now notice all of the input text boxes on this form. Notice that they are bound to our **SupportForm** fields. `th:field` on line 27 is bound to `id`. On line 31, we have the field bound to `name` and on line 35, we have the field bound to the `email` member variable of the **SupportForm**.

```
<input type="text" th:field="*{id}"/>
<input type="text" th:field="*{name}"/>
<input type="text" th:field="*{email}"/>
```

SupportForm.java

```
3 public class SupportForm {
4
5     private Long id;
6     private String name;
7     private String email;
8     private String gender;
9     private String membership;
10    private String content;
11    public Long getId() {
12        return id;
13    }
14    public void setId(Long id) {
15        this.id = id;
16    }
17 }
```

The **thymeleaf** template engine and spring will use *reflection* to bind these form fields to the right member variables of our *binding model object*, the **SupportForm**.

Observe that we have specified the gender field in the form of a radio button. We have two radio buttons, one for Male, one for Female.

```
<input type="radio" th:field="*{gender}" value="Male" />Male
<input type="radio" th:field="*{gender}" value="Female" />Female
```

Both of these radio buttons are bound to the `gender` member variable.

The **membership** field is filled in using a drop down that is the select HTML element. It's bound to the membership variable in the **SupportForm** object.

```
<select th:field="*{membership}">
```

We then specify the options using **th:each**.

```
<option th:each="ListItem : ${membershipList}" th:value="${ListItem}"
th:text="${ListItem}" />
```

The **th:each** construct in **thymeleaf** basically iterates over all of the items in the membershipList that we had sent down. For every **listItem**, we display the contents of the **listItem** as a part of the *drop down menu*. Both **th:value** and **th:text** are set to the **listItem** within our **membershipList**. These will be Platinum, Gold and Silver.

```
14 @Controller
15 public class SupportFormController {
16
17     @GetMapping("/support")
18     public String complaintForm(Model model) {
19         model.addAttribute("supportDetails", new SupportForm());
20
21         List<String> membershipList = Arrays.asList("Platinum", "Gold", "Silver");
22         model.addAttribute("membershipList", membershipList);
23
24         return "support";
25     }
26 }
```

And finally, the actual customer support request is bound to the content field in the **SupportForm**.

```
<input type="text" th:field="*{content}" />
```

And finally, we have Submit buttons and Reset buttons for our form. Submit will make a post request to our post handler.

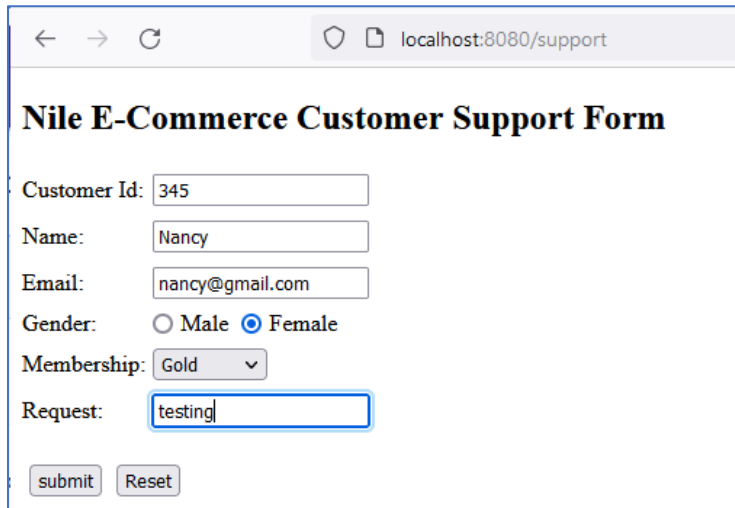
There's one last file that we need to look at, the **submit.html**.

```
submit.html
1 <!DOCTYPE html>
2 <html xmlns:th="https://www.thyLeaf.org">
3 <head>
4 <meta charset="ISO-8859-1">
5 <title>Support Request Submitted</title>
6 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7 </head>
8 <body>
9 <h2>Support request submitted successfully</h2>
10 <p th:text="Customer Id: ' + ${supportDetails.id}" />
11 <p th:text="Name: ' + ${supportDetails.name}" />
12 <p th:text="Email: ' + ${supportDetails.email}" />
13 <p th:text="Gender: ' + ${supportDetails.gender}" />
14 <p th:text="Membership: ' + ${supportDetails.membership}" />
15 <p th:text="Content: ' + ${supportDetails.content}" />
16
17 <a href="/support">Submit another?</a>
18 </body>
19 </html>
```

This is the view that'll be rendered once we submit the form. This simply uses **thymeleaf** to print out all of the submission details in our SupportForm, Customer Id, Name, Email, Gender, Membership and Content.

12. Run and Test The Application

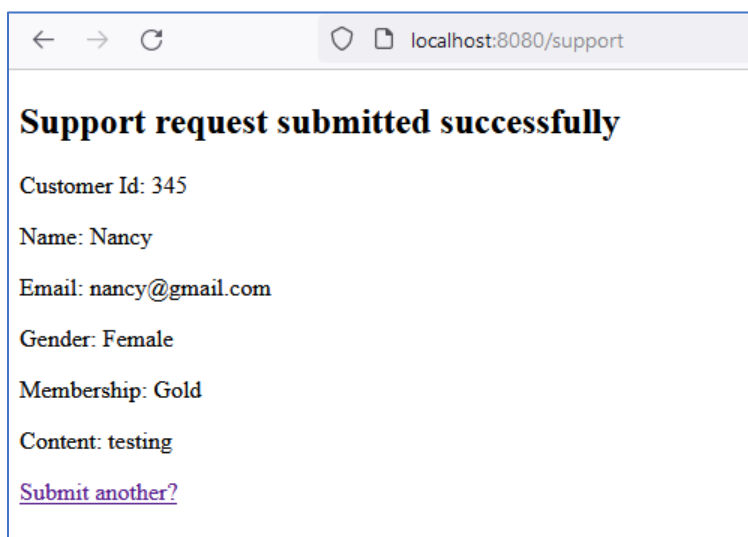
Now it's time for us to run this code and see how our form works. Head over to your browser and go to <http://localhost:8080/support>.



A screenshot of a web browser window showing a form titled "Nile E-Commerce Customer Support Form". The browser's address bar shows "localhost:8080/support". The form contains the following fields and controls:

- Customer Id:
- Name:
- Email:
- Gender: ☐ Male ☒ Female
- Membership:
- Request:
- Buttons:

This will render our E-Commerce Support Form. Just fill in details on this form. Notice that the membership list is a drop down with Platinum, Gold and Silver. Select one of these options and fill in the remaining details. So once you hit Submit, we'll be redirected to the Submit page where we simply print out the details of our submission.



A screenshot of a web browser window showing a confirmation page titled "Support request submitted successfully". The browser's address bar shows "localhost:8080/support". The page displays the submitted details:

- Customer Id: 345
- Name: Nancy
- Email: nancy@gmail.com
- Gender: Female
- Membership: Gold
- Content: testing
- A link: [Submit another?](#)

When we hit Submit on our form, a post request to the **/support** path is made to our controller. And this **submitComplaint** handler will handle that request.

```
14 @Controller
15 public class SupportFormController {
16
17     public String complaintForm(Model model) {}
18
19     @PostMapping("/support")
20     public String submitComplaint(@ModelAttribute SupportForm supportDetails, Model model) {
21         model.addAttribute("supportDetails", supportDetails);
22         return "submit";
23     }
24 }
```

The information that the user specified, will be available in the **SupportForm** injected into this method. This is thanks to the **@ModelAttribute** annotation. We then add this **SupportForm** to the Model once again and render the Submit page which is what you see here on screen. And the Submit page renders the support details.

Performing Form Validation

In the previous demo, we set up a form to accept user input. But we didn't really validate the contents of the form when the user submitted the form to the back end. Let's fix this here in this demo.

13. Add new dependency for Validation in POM.xml

I need to add an additional dependency for form validation within my pom.xml file. This is the **spring-boot-starter-validation** dependency descriptor.



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.5.7</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.mytutorial</groupId>
12  <artifactId>springboot</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>springboot</name>
15  <description>Demo project for Spring Boot</description>
16  <properties>
17    <java.version>1.8</java.version>
18  </properties>
19  <dependencies>
20    <dependency>
21      <groupId>org.springframework.boot</groupId>
22      <artifactId>spring-boot-starter-thymeleaf</artifactId>
23    </dependency>
24    <dependency>
25      <groupId>org.springframework.boot</groupId>
26      <artifactId>spring-boot-starter-web</artifactId>
27    </dependency>
28    <dependency>
29      <groupId>org.springframework.boot</groupId>
30      <artifactId>spring-boot-starter-validation</artifactId>
31    </dependency>
32  </dependencies>
33  <dependency>
34    <groupId>org.springframework.boot</groupId>
35    <artifactId>spring-boot-starter-test</artifactId>
36    <scope>test</scope>
37  </dependency>
38 </dependencies>
39
40 <build>
41   <plugins>
42     <plugin>
43       <groupId>org.springframework.boot</groupId>
44       <artifactId>spring-boot-maven-plugin</artifactId>
45     </plugin>
46   </plugins>
47 </build>
48
49 </project>
```

This starter dependency in Spring Boot allows us to use the JavaX validation API and the Hibernate validator.

14. Include Validation on each property in Model Object

You can think of this demo as a continuation of the previous one. I won't show you the code that remains the same, we'll only focus on what's changed. And what's changed is our model object, the **SupportForm** object.

```
SupportForm.java
1 package com.mytutorial.springboot.model;
2
3 import javax.validation.constraints.Email;
4 import javax.validation.constraints.Min;
5 import javax.validation.constraints.NotEmpty;
6 import javax.validation.constraints.NotNull;
7 import javax.validation.constraints.Pattern;
8 import javax.validation.constraints.Size;
9
10 public class SupportForm {
11
12     @NotNull
13     @Min(value = 1000, message = "customer ID should be >= 10000")
14     private Long id;
15
16     @NotNull
17     @Size(min = 5, max = 50)
18     private String name;
19
20     @NotEmpty
21     @Email
22     private String email;
23
24     @NotNull
25     private String gender;
26
27     @NotNull
28     private String membership;
29
30     @NotNull
31     @Pattern(regexp = "[a-zA-Z0-9 ]{3,255}", message = "please enter only letters and numbers")
32     private String content;
33
34     public Long getId() {
35         return id;
36     }
37     public void setId(Long id) {
38         this.id = id;
39     }
40     public String getName() {
41         return name;
42     }
43     public void setName(String name) {
```

Notice that we still continue to have the same member variables, but we've added additional annotations to each of these member variables.

All of these annotations are validations that we want performed on these individual fields. And all of these annotations are part of the **javax.validation** namespace. Let's look at each of them in turn.

- The **@NotNull** annotation, which is applied to all of the member variables of this form indicates that that form field cannot have a null value. It has to have some value.
- The **@NotEmpty** annotation checks for everything that **@NotNull** does, that is the field should not have a null value. In addition, it checks that the length of *the field is non-zero*, that is, the email string has at least one character.

- The **@Min** specifies the *minimum* value for this number, example `@Min(value = 1000)` the minimum value is 10,000. If this is not satisfied, the message that'll be shown to the user is "customer ID should be `>= 10000`". The message parameter in the validation annotation allows us to configure custom error messages.
- The **@Size** annotation which specifies a *minimum* and *maximum* size for the name field. Based on requirements `@Size(min = 5, max = 50)`, the minimum number of characters for the name should be 5 and the maximum 50.
- The **@Email** validation checks to see that we have a well formed email in this field.
- The **@Pattern** validation annotation. That allows us to specify a regular expression, which the text in our content field needs to adhere to. The regular expression `@Pattern(regex = "[a-zA-Z0-9] {3,255}")` basically says that we can only have letters, numbers and blank spaces in this content field. And we should have no more than 255 characters, between 3 and 255 characters.

15. Update routing Controller when handling invalid validation

Another change that we need to make here is after adding validation to our `SupportForm` object, we need to reject forms which are not valid. If we go to the **SupportFormController**, the original get request which renders the form remains exactly the same.

```

1 package com.mytutorial.springboot.controller;
2
3 import java.util.Arrays;
4 import java.util.List;
5
6 import javax.validation.Valid;
7
8 import org.springframework.stereotype.Controller;
9 import org.springframework.ui.Model;
10 import org.springframework.validation.BindingResult;
11 import org.springframework.web.bind.annotation.GetMapping;
12 import org.springframework.web.bind.annotation.ModelAttribute;
13 import org.springframework.web.bind.annotation.PostMapping;
14
15 import com.mytutorial.springboot.model.SupportForm;
16
17 @Controller
18 public class SupportFormController {
19
20     @GetMapping("/support")
21     public String complaintForm(Model model) {
22         model.addAttribute("supportDetails", new SupportForm());
23
24         List<String> membershipList = Arrays.asList("Platinum", "Gold", "Silver");
25         model.addAttribute("membershipList", membershipList);
26
27         return "support";
28     }
29
30     @PostMapping("/support")
31     public String submitComplaint(@Valid @ModelAttribute("supportDetails") SupportForm supportDetails,
32                                 BindingResult bindingResult,
33                                 Model model) {
34
35         if (bindingResult.hasErrors()) {
36             List<String> membershipList = Arrays.asList("Platinum", "Gold", "Silver");
37             model.addAttribute("membershipList", membershipList);
38
39             System.err.println("Validation not pass");
40             return "support";
41         }
42         System.out.println("no error, validation pass");
43
44         model.addAttribute("supportDetails", supportDetails);
45         return "submit";
46     }
47 }

```

There is no change in the code for this first method `complaintForm`. But if you scroll down, the **submitComplaint** method is now a little different.

Notice that it still takes in an input argument, an object of the type `SupportForm`. This input argument has two annotations,

- The **@ModelAttribute** for the `supportDetails`. This tells Spring that this `SupportForm` object bound to our form user interface should be injected into this method which handles form submission.
- The additional annotation that we have here is the **@Valid** annotation. This tells Spring that we have specific validation constraints in the form fields and only valid forms should be accepted.

So the SupportForm fields will be validated and any errors will be added to the binding result which is a second input argument.

Now it's important for you to note here that the binding result input argument should follow the model object that has been bound to our form. Otherwise, this won't work.

And finally, we have the model for our MVC application, that is a third input argument. We've made a couple of changes here to the body of this method. We've instantiated the membershipList, platinum, gold and silver and added that as an attribute to the model.

This will be needed by the UI which re-renders our form if the form is found to be invalid. We then check to see if the submitted form had errors by using **bindingResult.hasErrors**. If this is true, we simply re-render the SupportForm once again.

By re-rendering the SupportForm, any errors associated with the fields of the form will be rendered in the UI. As you shall see when we take a look at the user interface in the **support.html**. If there are no binding result errors, we simply render the submit view.

We know that the submit view uses the SupportForm to show us our submission details. But we haven't explicitly added the support form as an object to our model. It's not really needed. TheSupportForm is tagged using @ModelAttribute, it will be automatically added to the model when we render the Submit view.

16. Update Entry View to include Validation Tags

Now, let's head over to the **support.html** page, which has a few changes to display errors in our form.

```

23 <h2> Nile E-Commerce Customer Support Form</h2>
24 <form action="#" th:action="@{/support}" th:object="${supportDetails}" method="post">
25   <div>
26     <label>Customer Id:</label>
27     <input type="text" th:field="*{id}" />
28     <span th:if="${#fields.hasErrors('id')}" th:errors="*{id}"></span>
29   </div>
30   <div>
31     <label>Name:</label>
32     <input type="text" th:field="*{name}" />
33     <span th:if="${#fields.hasErrors('name')}" th:errors="*{name}"></span>
34   </div>
35   <div>
36     <label>Email:</label>
37     <input type="text" th:field="*{email}" />
38     <span th:if="${#fields.hasErrors('email')}" th:errors="*{email}"></span>
39   </div>
40   <div>
41     <label>Gender:</label>
42     <input type="radio" th:field="*{gender}" value="Male" />Male
43     <input type="radio" th:field="*{gender}" value="Female" />Female
44     <span th:if="${#fields.hasErrors('gender')}" th:errors="*{gender}"></span>
45   </div>
46   <div>
47     <label>Membership:</label>
48     <select th:field="*{membership}">
49       <option th:each="listItem : ${membershipList}" th:value="${listItem}" th:text="${listItem}" />
50     </select>
51     <span th:if="${#fields.hasErrors('membership')}" th:errors="*{membership}"></span>
52   </div>
53   <div>
54     <label>Request:</label>
55     <input type="text" th:field="*{content}" />
56     <span th:if="${#fields.hasErrors('content')}" th:errors="*{content}"></span>
57   </div>
58   <p><input type="submit" value="submit" /><input type="reset" value="Reset" /></p>
59 </form>

```

There is absolutely no change to the form HTML element, the **th:object** still binds to the supportDetails form.

The method is post and the action is /support. Observe on lines 28, 33 and 38, we have a span HTML element which is rendered conditionally using th:if.

```
<span th:if="${#fields.hasErrors('id')}}" th:errors="*{id}"></span>
<span th:if="${#fields.hasErrors('name')}}" th:errors="*{name}"></span>
<span th:if="${#fields.hasErrors('email')}}" th:errors="*{email}"></span>
<span th:if="${#fields.hasErrors('gender')}}" th:errors="*{gender}"></span>
<span th:if="${#fields.hasErrors('membership')}}"
th:errors="*{membership}"></span>
<span th:if="${#fields.hasErrors('content')}}" th:errors="*{content}"></span>
```

The contents of the span element will be displayed only if that associated field has errors. So on line 28, if field.hasErrors(id) there are errors corresponding to the ID field will be displayed. On line 33, if the name field has errors, those errors will be displayed.

And on line 38, if the email field has errors, those errors will be displayed.

In exactly the same way, we've set up span elements to display the errors in the other form fields as well. The gender field has a span element on line 44.

The membership field has a span element on line 51. And finally, the request field has a span element on line 56.

It's now time to see our validated form in action. Run this code and let's head over to local host 8080/support.

Here is our form, and I'm going to hit Submit without filling in anything. And you can see that all of our fields display errors except for the Membership dropdown.

Nile E-Commerce Customer Support Form

Customer Id: must not be null

Name: size must be between 5 and 50

Email: must not be empty

Gender: ☐ Male ☐ Female must not be null

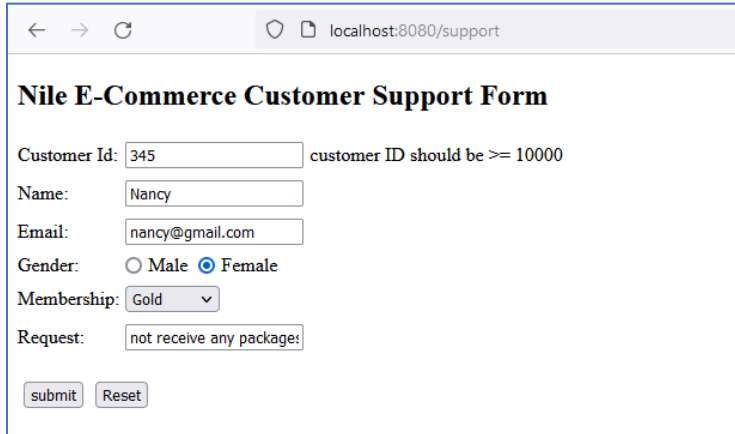
Membership:

Request: please enter only letters and numbers

The error corresponding to Customer ID says must not be null. The error corresponding to Email says must not be empty. The error corresponding to Request says please enter just letters and numbers. I'm now going to fill up a few details on this form. Let's see if we can hit some of the more specific errors. I'm going to hit Submit.

See if you can figure out what error will show up. You can see that the customer ID is 345, but it should be greater than, equal to 10,000. This is thanks to the @Min validation annotation on the ID field. I'm going to fill up this form once again, this time with a few different values, and let's

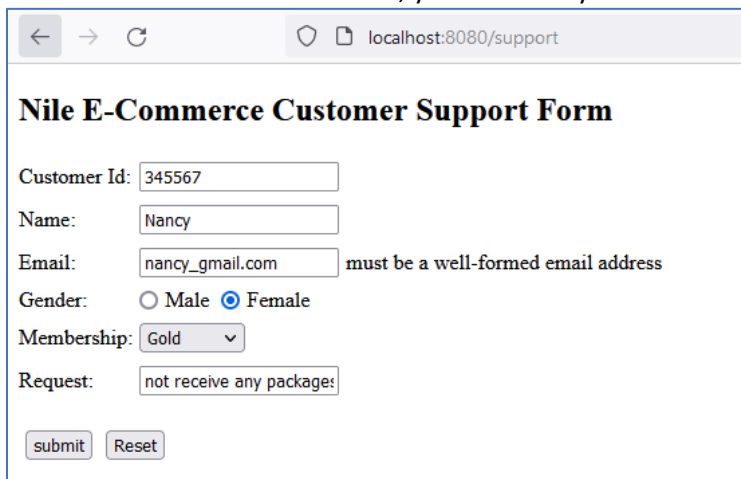
see what validation constraint is hit. I'm going to hit Submit here. And here is an error message, says that the size of the Name field should be between 5 and 50.



A screenshot of a web browser displaying the 'Nile E-Commerce Customer Support Form' at localhost:8080/support. The form contains several fields: 'Customer Id' with value '345' and an error message 'customer ID should be >= 10000'; 'Name' with value 'Nancy'; 'Email' with value 'nancy@gmail.com'; 'Gender' with 'Female' selected; 'Membership' with a dropdown set to 'Gold'; and 'Request' with value 'not receive any package!'. At the bottom are 'submit' and 'Reset' buttons.

This is thanks to the @Size annotation on the Name member variable. Let's try this once again, I'm going to fill up details on this form once again and this time we'll hit a different error.

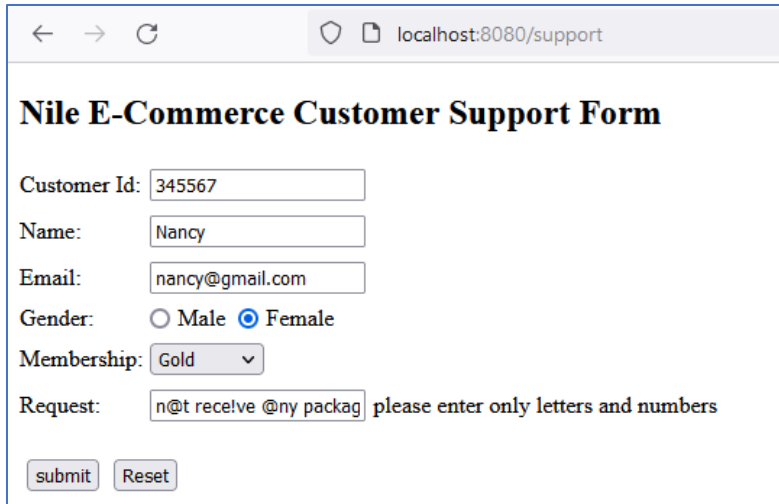
Let's now hit Submit on this form, you can clearly see that the email address is not well formed.



A screenshot of the same web browser displaying the 'Nile E-Commerce Customer Support Form'. The 'Customer Id' field now has the value '345567'. The 'Email' field has the value 'nancy_gmail.com' and shows an error message 'must be a well-formed email address'. The other fields and buttons remain the same as in the previous screenshot.

This error that you see here is because of the @Email annotation on the email member variable in the SupportForm.

Let's try this once again. I hit refresh and I'm going to fill up the details on this form once again. This time I have some special characters within my request field.

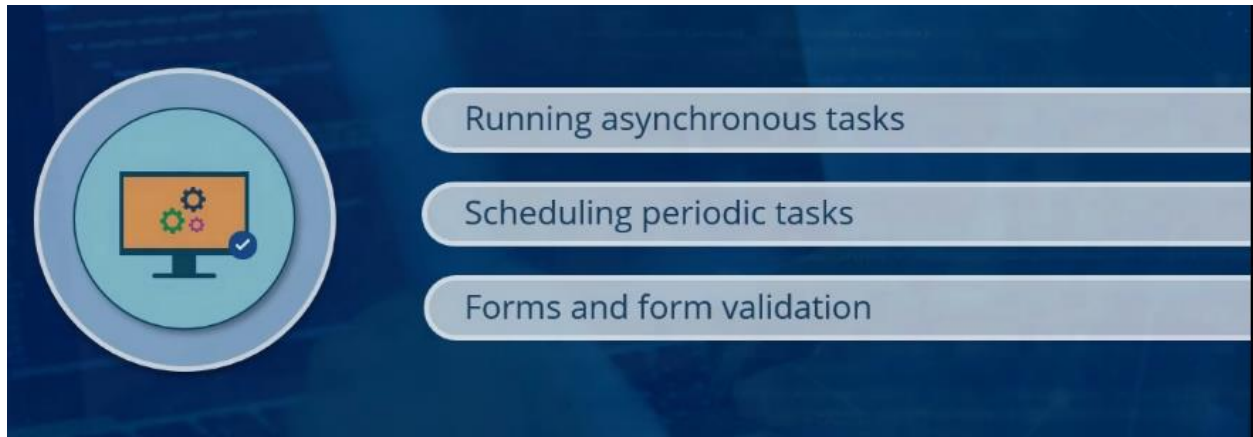


The screenshot shows a web browser window with the address bar displaying 'localhost:8080/support'. The page title is 'Nile E-Commerce Customer Support Form'. The form contains the following fields and controls:

- Customer Id:** A text input field containing '345567'.
- Name:** A text input field containing 'Nancy'.
- Email:** A text input field containing 'nancy@gmail.com'.
- Gender:** Radio buttons for 'Male' and 'Female', with 'Female' selected.
- Membership:** A dropdown menu showing 'Gold'.
- Request:** A text input field containing 'n@t receive @ny packag'. To the right of the input is a validation message: 'please enter only letters and numbers'.
- Buttons:** 'submit' and 'Reset' buttons at the bottom left.

Now the pattern that we have specified does not allow special characters, that's why we see the message, please enter only letters and numbers.

Course Summary



In this course, we saw how we could work with asynchronous methods, scheduled periodic tasks, configure handler mappings, and use forms from within Spring Boot. We started off by configuring our Spring Boot application such that it could run tasks asynchronously on a background thread. We configured the executor for the tasks and also how many threads should be part of the thread pool. We also saw how Spring Boot allows us to schedule periodic tasks which run either at periodic intervals or with a fixed delay. We understood in detail how the spring scheduler can be configured for our specific use case. Finally, we explored how we could access a request parameter set by the user using the add request param annotation.

And how dynamic URL path elements could be extracted using at path variable annotations. We also set up a form using the Thymeleaf template engine, and used spring annotations to validate the form fields and display errors.

Quiz

1. Which annotations are needed for your application when you need to run scheduled tasks asynchronously?
@EnableTimer
@EnableAsyncScheduling
✓ **@EnableAsync**
✓ **@EnableScheduling**
2. What Thymeleaf attribute is specified on a form to bind a form to a Java object?
✓ **th:object**
th:action
th:text
th:value
3. Which of the annotations validates the submitted form?
@ValidateForm
@ModelAttribute
✓ **@Valid**
@RequestParam
4. To set up your Spring bean to be executed, it needs to implement which interface?
Runner
Runnable
✓ **CommandLineRunner**
Executor
5. Which annotation allows us to extract elements of a dynamic URL?
@RequestParam
✓ **@PathVariable**
@GetMapping
@PostMapping
6. Which configurations of the `@Scheduled` annotation invokes the method 4 seconds after the completion of the previous invocation of the method?
@Scheduled(initialDelay = 4000)
@Scheduled(fixedRate = 4000)
@Scheduled(gap = 4000)
✓ **@Scheduled(fixedDelay = 4000)**