**Hands On Lab 1**

**Creating Asynchronous Methods**

# **Creating Asynchronous Methods**

This lab walks you through creating asynchronous queries to GitHub. The focus is on the asynchronous part, a feature often used when scaling services.

## What You Will build

You will build a lookup service that queries GitHub user information and retrieves data through GitHub’s API. One approach to scaling services is to run expensive jobs in the background and wait for the results by using Java’s [CompletableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html) interface. Java’s CompletableFuture is an evolution from the regular Future. It makes it easy to pipeline multiple asynchronous operations and merge them into a single asynchronous computation.

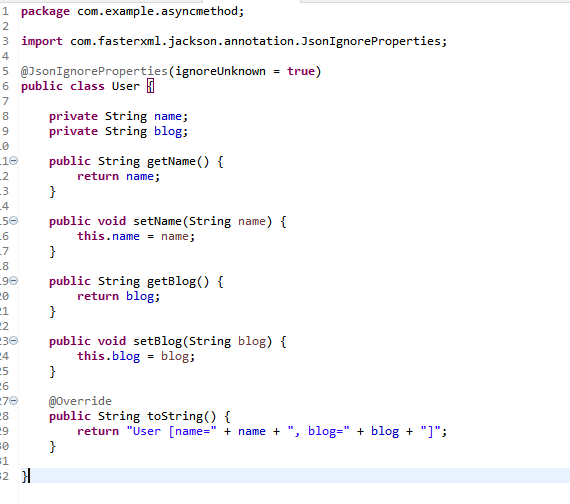
## Starting with Spring Initializr

1. Create a new Spring Starter project in Spring Tool Suite.
2. Name it ***asynch-method***.
3. For **Dependencies** , select **Spring Web**.

## Create a Representation of a GitHub User

Before you can create a GitHub lookup service, you need to define a representation for the data you will retrieve through GitHub’s API.

To model the user representation, create a resource representation class. To do so, provide a plain old Java object with fields, constructors, and accessors, as the following example (from src/main/java/com/example/asyncmethod/User.java) shows:



Spring uses the [Jackson JSON](https://wiki.fasterxml.com/JacksonHome) library to convert GitHub’s JSON response into a User object. The @JsonIgnoreProperties annotation tells Spring to ignore any attributes not listed in the class. This makes it easy to make REST calls and produce domain objects.

In this lab, we grab only the name and the blog URL for demonstration purposes.

## Create a GitHub Lookup Service

Next, you need to create a service that queries GitHub to find user information. The following listing (from src/main/java/com/example/asyncmethod/GitHubLookupService.java) shows how to do so:



The GitHubLookupService class uses Spring’s RestTemplate to invoke a remote REST point (api.github.com/users/) and then convert the answer into a User object. Spring Boot automatically provides a RestTemplateBuilder that customizes the defaults with any auto-configuration bits (that is, MessageConverter).

The class is marked with the @Service annotation, making it a candidate for Spring’s component scanning to detect and add to the application context.

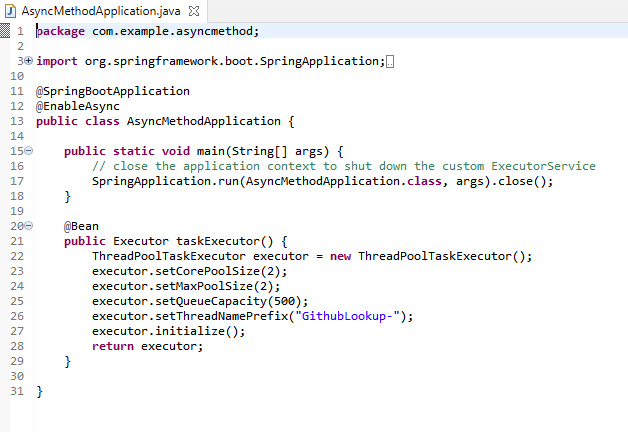
The findUser method is flagged with Spring’s @Async annotation, indicating that it should run on a separate thread. The method’s return type is [CompletableFuture<User>](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html) instead of User, a requirement for any asynchronous service. This code uses the completedFuture method to return a CompletableFuture instance that is already completed with result of the GitHub query.

|  |  |
| --- | --- |
|  | Creating a local instance of the GitHubLookupService class does NOT allow the findUser method to run asynchronously. It must be created inside a @Configuration class or picked up by @ComponentScan. |

The timing for GitHub’s API can vary. To demonstrate the benefits later in this lab, an extra delay of one second has been added to this service.

## Make the Application Executable

To run a sample, you can create an executable jar. Spring’s @Async annotation works with web applications, but you need not set up a web container to see its benefits. The following listing (from src/main/java/com/example/asyncmethod/AsyncMethodApplication.java) shows how to do so:



@SpringBootApplication is a convenience annotation that adds all of the following:

@Configuration: Tags the class as a source of bean definitions for the application context.

@EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if spring-webmvc is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet.

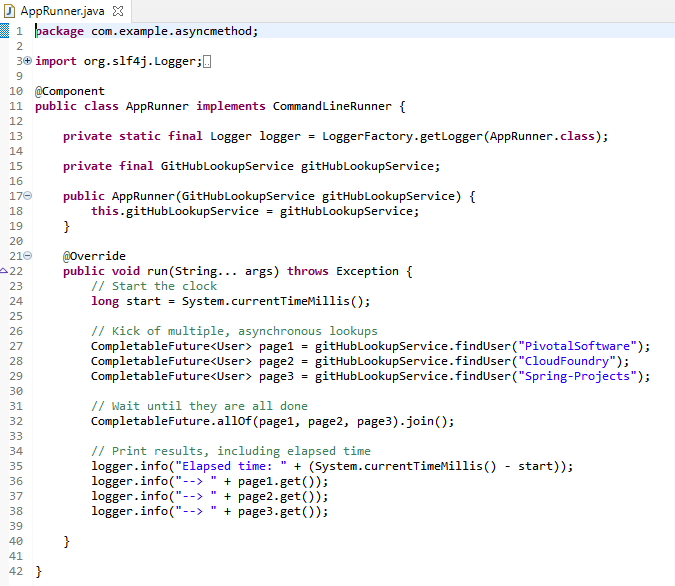
@ComponentScan: Tells Spring to look for other components, configurations, and services in the com/example package, letting it find the controllers.

The main() method uses Spring Boot’s SpringApplication.run() method to launch an application. Did you notice that there was not a single line of XML? There is no web.xml file, either. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

The [@EnableAsync](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/scheduling.html" \l "scheduling-annotation-support) annotation switches on Spring’s ability to run @Async methods in a background thread pool. This class also customizes the Executor by defining a new bean. Here, the method is named taskExecutor, since this is the [specific method name for which Spring searches](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/EnableAsync.html). In our case, we want to limit the number of concurrent threads to two and limit the size of the queue to 500. There are [many more things you can tune](https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/integration.html" \l "scheduling-task-executor). If you do not define an Executor bean, Spring creates a SimpleAsyncTaskExecutor and uses that.

There is also a [CommandLineRunner](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/" \l "boot-features-command-line-runner) that injects the GitHubLookupService and calls that service three times to demonstrate the method is executed asynchronously.

You also need a class to run the application. Create that in src/main/java/com/example/asyncmethod/AppRunner.java. The following listing shows that class:



### Build an executable JAR

You can run the application from the command line with Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you use Maven, you can run the application by using ./mvnw spring-boot:run. Alternatively, you can build the JAR file with ./mvnw clean package and then run the JAR file, as follows:

java -jar target/gs-async-method-0.1.0.jar

|  |  |
| --- | --- |
|  | The steps described here create a runnable JAR. You can also [build a classic WAR file](https://spring.io/guides/gs/convert-jar-to-war/). |

Of course, you can always run it from the Boot Dashboard in Spring Tool Suite.

The application shows logging output, showing each query to GitHub. With the help of the allOf factory method, we create an array of CompletableFuture objects. By calling the join method, it is possible to wait for the completion of all of the CompletableFuture objects.

The following listing shows typical output from this sample application:

2016-09-01 10:25:21.295 INFO 17893 --- [ GithubLookup-2] hello.GitHubLookupService : Looking up CloudFoundry

2016-09-01 10:25:21.295 INFO 17893 --- [ GithubLookup-1] hello.GitHubLookupService : Looking up PivotalSoftware

2016-09-01 10:25:23.142 INFO 17893 --- [ GithubLookup-1] hello.GitHubLookupService : Looking up Spring-Projects

2016-09-01 10:25:24.281 INFO 17893 --- [ main] hello.AppRunner : Elapsed time: 2994

2016-09-01 10:25:24.282 INFO 17893 --- [ main] hello.AppRunner : --> User [name=Pivotal Software, Inc., blog=https://pivotal.io]

2016-09-01 10:25:24.282 INFO 17893 --- [ main] hello.AppRunner : --> User [name=Cloud Foundry, blog=https://www.cloudfoundry.org/]

2016-09-01 10:25:24.282 INFO 17893 --- [ main] hello.AppRunner : --> User [name=Spring, blog=https://spring.io/projects]

Note that the first two calls happen in separate threads (GithubLookup-2, GithubLookup-1) and the third one is parked until one of the two threads became available. To compare how long this takes without the asynchronous feature, try commenting out the @Async annotation and runing the service again. The total elapsed time should increase noticeably, because each query takes at least a second. You can also tune the Executor to increase the corePoolSize attribute for instance.

Essentially, the longer the task takes and the more tasks are invoked simultaneously, the more benefit you see from making things asynchronous. The trade off is handling the CompletableFuture interface. It adds a layer of indirection, because you are no longer dealing directly with the results.

## Summary

Congratulations! You have just developed an asynchronous service that lets you scale multiple calls at once.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*