Developing Applications with Spring Boot and Spring Cloud

Introduction

Prerequisites

Knowledge to have beforehand:

- Spring Framework/Core
- Spring Web

Your local environment:

- Java - make sure you download and install the **JDK** instead of JRE.
- Spring Tool Suite (download STS)
    - ensure that the IDE is using the latest installed version of Maven
        - Preferences > Maven > Installations
- Gradle
- Optional: JSON formatter in your browser

High Level Objectives

- Gain proficiency with the following aspects of Spring IO:
    - Spring Boot
    - Spring REST Services
    - Spring Cloud
        - Config Server
        - Eureka
        - Hystrix

A complete, working version of the demo app can be cloned from here:
https://github.com/davejonesstl/spring-boot-demo

Ports

Demo App	9000
Spring Cloud Config Server	8888
Spring Cloud Netflix Eureka Server	8761
Spring Netflix Hystrix Dashboard	7979
Fortune Teller Service App	8080

Labs

Lab 1 : Spring Boot

Objectives

- Learn how Spring Boot works by creating a simple Spring Boot application
- Build a REST service
- Learn about Spring Boot Actuator

1. Use Spring Initializer to generate the project
    a. Go to Spring Initializer site.
    b. Click 'switch to full version' (link at bottom of page)
    c. Generate a **[gradle project]** with **[<latest stable version>]**
    d. Group = com.demo
    e. Artifact = demo
    f. Packaging = Jar
    g. Java Version = 1.8
    h. Language = Java
    i. Dependencies -> Search for "**Web**" and only add that one.
    j. Click "Generate Project". This will generate and download a zip containing the skeleton of this demo app.
    k. Unzip it
2. Import the demo app into STS
    a. Start STS
    b. File -> Import -> Gradle Project
    c. Select the folder you unzipped from Spring Initializer. When the import is finished, you should see your "demo" project in the STS workspace
3. Run the "build" Gradle task to ensure it works
4. Implement the HelloController
    a. Create a new Java class called com.demo.HelloController
    b. Paste this code in for the implementation:

```
package com.demo;

import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
public class HelloController {

    @RequestMapping("/")
```

```
            public String index() {
                return "Greetings from KC Spring Boot demo!";
            }

        }
```

5. Build and launch the app by running the Gradle task “bootRun”
6. Test the app
    a. Open browser to http://localhost:8080 and verify you get the message “Greetings from KC Spring Boot demo!”
7. Add Actuator for production-grade services
    a. Open gradle.build and add this line to the dependencies section:

    ```
    compile("org.springframework.boot:spring-boot-starter-actuator")
    ```

    b. Look at the health report: http://localhost:8080/health

Your Spring Boot app with built-in health monitoring is DONE (for now)!

Lab 2: Add a RESTful service

This lab will show you how to do two things:
1. Extend your HelloController to be a RESTful service when invoked with /hello-world.
2. Change the ports on which your app and Actuator are listening

This is based off of the Building a RESTful Web Service with Spring Boot Actuator guide from Spring.

1. Implement Greeting class to represent your response data.

```
package com.demo;

public class Greeting {

    private final long id;
    private final String content;

    public Greeting(long id, String content) {
        this.id = id;
        this.content = content;
    }

    public long getId() {
        return id;
```

```
        }

        public String getContent() {
            return content;
        }

    }
```

2. Add the `counter` instance variable and the `sayHello` method to your HelloController

```
private final AtomicLong counter = new AtomicLong();

@RequestMapping(value="/hello-world", method=RequestMethod.GET)
public @ResponseBody Greeting sayHello(@RequestParam(value="name", required=false,
defaultValue="Stranger") String name) {
    return new Greeting(counter.incrementAndGet(), String.format(template, name));
}
```

3. Switch to a different server port by creating an application.yml file in /src/main/resources, and use this as the content of the file:

```
server:
  port: 9000
management:
  port: 9001
  address: 127.0.0.1
```

4. Restart the app and verify the two new endpoints are working:
   a. http://localhost:9000/hello-world
   b. http://localhost:9001/health

Resources

- Spring Boot Reference
  - Actuator section
- Spring Framework Reference
- Spring Boot presentation from SpringOne 2014

Scan the Spring Boot reference docs. These cover A LOT more of what can be done with Spring Boot. At a minimum, familiarize yourself with the TOC so that you are familiar with the reference and review the Actuator section.

Lab Prep: Fortune-teller Setup

For the remainder of the labs we will leverage the fortune-teller sample from this github repo in addition to the 'demo' spring boot application that we have created. Though it is a fully completed application we will be using our existing 'demo' application as part of the lab so that we have some additional hands-on experience and not just a review of fully implemented code.

We have a copy of the fully built and packaged fortune-teller app. Alternatively, you can pull it down and build it yourself.

Build it Yourself (in STS or if you have maven installed)

1. git clone https://github.com/spring-cloud-samples/fortune-teller.git
2. cd fortune-teller
3. mvn clean install

Copy Pre-build fortune-teller application

- ~~Clone/download this repo to wherever you like (<install location>)~~
- Unzip the fortune teller application
- ~~*Optional*: import as new maven project to Spring Tool Suite (it may take a moment to pull of the dependencies) in order to easily view the example project code~~

Run the Config Server

```
cd <install location>/fortune-teller-config-server/target
java -jar fortune-teller-config-server-0.0.1-SNAPSHOT.jar
```

Lab 3: Spring Cloud Netflix:Eureka

Objectives

- Learn about the Eureka Server
- Build out the demo application to use the Eureka Server to discover a service

Lesson

- Scan the References docs section on Eureka
- Ensure Spring Cloud Config Server is still running: http://localhost:8888/mappings

- Start Eureka jar

```
        cd <install location>/fortune-teller-eureka/target
        java -jar fortune-teller-eureka-0.0.1-SNAPSHOT.jar
```

- Start fortune teller service jar

```
        cd <install location>/fortune-teller-fortune-service/target
        java -jar fortune-teller-eureka-0.0.1-SNAPSHOT.jar
```

- Modify “Demo Application” to consume the fortune teller service using eureka registered urls and display the fortunes.
- Add Netflix and Eureka to build.gradle

```
dependencyManagement {
  imports {
    mavenBom 'org.springframework.cloud:spring-cloud-dependencies:Brixton.SR6'
    mavenBom 'org.springframework.cloud:spring-cloud-netflix:1.2.0.RELEASE'
  }
}

dependencies {
        compile('org.springframework.boot:spring-boot-starter-web')
        testCompile('org.springframework.boot:spring-boot-starter-test')
        compile('org.springframework.boot:spring-boot-starter-actuator')
        compile('org.springframework.cloud:spring-cloud-starter-eureka')
}
```

- Enable the DemoApp to consume eureka server using “@EnableEurekaClient”
- Inject an instance of DiscoveryClient class to your controller
- Create a new controller method to retrieve the fortunes
    - Use the discoveryClient object in your controller method to retrieve an instance of ServiceInstance with information to the FORTUNES eureka registered server.
    - retrieve the Uri of the server from the located ServerInstance objects
    - Consume the /random endpoint of the registered url in eureka using a RestTemplate spring object.

```
    @Autowired
    private DiscoveryClient discoveryClient;

    @RequestMapping("/fortune")
    public Fortune getFortune() throws Exception {
        URI fortuneURI = getServiceUrl();
        String fortuneURIFull = fortuneURI + "/random";
        RestTemplate restTemplate = new RestTemplate();
        Fortune fortune = restTemplate.getForObject(fortuneURIFull, Fortune.class);
        return fortune;
    }

    public URI getServiceUrl() throws Exception {
        List<ServiceInstance> list = discoveryClient.getInstances("fortunes");
        if (list == null || list.size() == 0 ) {
            throw new Exception("No service instances found!");
        }
        return list.get(0).getUri();
    }
```

- Create the Fortune domain object:

```
public class Fortune {

    private Long id;
    private String text;

    public Long getId() {
        return id;
    }

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

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }
}
```

- Restart the demo app (gradle bootRun) and test at http://localhost:9000/fortunes
- The fortune teller service will return fortunes in json representation as follow "{'id':1','text':'text of the quote'}".

Resources

- Spring Cloud project page
- Spring Cloud Netflix Reference
- Spring Cloud Netflix project page
- More sample projects on this page and this repo

Lab 4: Spring Cloud Netflix: Hystrix

Objectives

- Learn about Hystrix
- Augment the demo application to include a Hystrix-based circuit break

Lesson

- Scan the References docs section on Hystrix
- Start hystrix-dashboard jar

```
cd <install location>/fortune-teller-hystrix-dashboard/target
java -jar fortune-teller-hystrix-dashboard-0.0.1-SNAPSHOT.jar
```

- Modify “Demo Application” to enable hystrix circuit breaker functionality
    - Add Hystrix starter to build.gradle
      ```
      dependencies {
              compile('org.springframework.boot:spring-boot-starter-web')
              testCompile('org.springframework.boot:spring-boot-starter-test')
              compile('org.springframework.boot:spring-boot-starter-actuator')
              compile('org.springframework.cloud:spring-cloud-starter-eureka')
              compile('org.springframework.cloud:spring-cloud-starter-hystrix')
      }
      ```
    - Enable your demo app to use Hystrix Circuit Breaker by adding `@EnableCircuitBreaker` annotation to your Spring Boot controller
    - ~~Make sure that the code for retrieving the fortunes developed in previous section is all wrapped in a spring component (It might cause problems if all the code is in the controller)~~
    - Annotate the `getFortune` method with @HystrixCommand and a fallbackMethod property.
      ```
              @HystrixCommand(fallbackMethod="getDefaultFortune")
      ```
    - Create the `getDefaultFortune` fallback method to return a default quote (the return types of the default method and the fallback method should be the same).
      ```
          public Fortune getDefaultFortune(){
             Fortune f = new Fortune();
             f.setId(999L);
             f.setText("Things not looking so great!");
             return f;
          }
      ```

- Bring the dashboard up in a browser window - http://localhost:7979
    - Search for http://localhost:9001/hystrix.stream
    - This stream is exposed by Actuator having been placed on the classpath
- Test the circuit
    - Relaunch your application and retrieve several fortunes
    - Bring down the fortune teller service
    - Try to retrieve more fortunes from the Demo app. This should trigger the quote from the fallback method to be displayed.
    - Reload the page quickly (20-30 times) and you should see the circuit open in the Hystrix dashboard. “Open” means the service is unavailable.
    - Relaunch the fortune-teller service and continue making calls.   You should see the service “heal” and make successful calls and the circuit close.

Resources

- More sample projects on this page and this repo

Additional Resources

- Spring projects
- Spring guides
- Spring reference docs

Lab 5: Spring Cloud Config

Objectives

- Learn about Spring Cloud Config
- Install a Spring Cloud Config Server
- Leverage configuration on the server from a client application

Lesson

- Scan the reference docs
- Ensure Config Server is still running. If not, start it.

```
cd <install location>/fortune-teller-config-server/target
java -jar fortune-teller-config-server-0.0.1-SNAPSHOT.jar
```

- Examine the configuration in the server via curl or browser

  http://localhost:8888/foo/development

- Configure the “demo app” to be a client of the Config Server
  - Add cloud and cloud config to build.gradle:

```
dependencies {
      compile('org.springframework.boot:spring-boot-starter-web')
      testCompile('org.springframework.boot:spring-boot-starter-test')
      compile('org.springframework.boot:spring-boot-starter-actuator')
      compile('org.springframework.cloud:spring-cloud-starter-eureka')
      compile('org.springframework.cloud:spring-cloud-starter-hystrix')
      compile('org.springframework.cloud:spring-cloud-starter-config')
}
```

  - Create a bootstrap.yml file in src/main/resources

```
spring:
  application:
    name: demo
  cloud:
    config:
      uri: http://localhost:8888
```

- Have the Hello controller display something from the Config Server
  - We use this specific key (‘fortune.fallbackFortune’) since this is the resource key (as seen in our examination of the config server resources) in the git repo we are using to externalize our configuration.

```
@Value("${fortune.fallbackFortune}")
String name = "World";
```

Resources

- Spring Cloud project page
- Spring Cloud Config Reference
- Spring Cloud Config project page
- More sample projects on this page and this repo