

Conclusion

Actuators, Metrics and Health Indicators Lab 🌣 \leq **Spring Boot** 8 of 45 lessons completed 18% 3m 🗀 **Agenda** MODULE 1 **Spring Boot Feature Introduction** MODULE 2 **Spring Boot - A Closer Look** MODULE 3 **Spring Boot - Spring Data JPA** MODULE 4 **Web Application with Spring Boot** MODULE 5 **RESTful Application with Spring Boot** MODULE 6 **Spring Boot Testing** MODULE 7 **Securing REST Application with Spring Security** MODULE 8 ^ **Actuator Actuators, Metrics and Health Indicators** 4m 🗀 **Setting up Actuators** 7m 🗀 **Metrics** 8m 🗀 **Health Indicators** 8m 🗀 **Actuators, Metrics and Health Indicators Lab** 45m 🖹 **Demo - Actuators, Metrics and Health Indicators Lab** 41m □3

囲

Actuators, Metrics and Health Indicators Lab

Purpose

In this lab you will gain experience with Spring Boot Actuator and its features.

Learning Outcomes

You will learn how to:

- Configure Spring Boot Actuator
- Expose some or all Actuator endpoints
- Define custom metrics
- Extend the /actuator/health endpoint to add custom health checks

You will be using the 44-actuator project.

Estimated time to complete: 45 minutes.

Prerequisites

The prerequisites are included as part of the Introduction to the Spring Professional Learning Path course Lab Setup lesson.

If you already completed it, you should be ready to do this lab. If not, assuming you already have JDK 11 or 17 installed and Java IDE, you will need to do following:

- 1. Download the lab codebase zip file.
- 2. Once you have downloaded the file, unzip it under a directory of your choice. The unzipped directory core-spring-labfiles/lab contains the lab projects.
- 3. Run ./mvnw clean install (if you plan on using Maven as your build tool).

Use Case

All production applications should have health monitoring, and will often need metric gathering. Actuator gives us both of these. You will be enabling actual your project and implementing mechanisms to provide an additional custom metric to indicate the number of times Account details have been requested. Additionally, you will be creating a custom HealthIndicator and adding that to the overall health status that is obtained.

Instructions

Enable Actuator

TODO-01: Check dependencies

1. In the pom.xml or build.gradle for the actuator project, look for TODO-01. We have added the dependency on the Spring Boot Ac

TODO-02: Review application

- 1. Run the application and using a browser that can display JSON (Firefox, Chrome), open http://localhost:8080/actuator/ and explore the links.
 - If you prefer you can use curl or Postman to examine the links referred to in this lab.
- 2. Now visit the http://localhost:8080/actuator/metrics endpoint.

Expose Actuator Endpoints

TODO-03 Expose HTTP actuator endpoints.

- 1. In the application properties file, expose the metrics and beans endpoints by setting the appropriate Spring Boot property.
 - These are two of the <u>many endpoints</u> that can be exposed.
- 2. Once the application restarts, visit the http://localhost:8080/actuator/metrics endpoint again. Now you will see a list of all the metrics tracked by Actu

Try fetching the data for one of the metrics by constructing a url based on the name of the metric. For example, visit http://localhost:8080/actuator/metrics/jym.memory.max.

3. View the beans our application has loaded by visiting http://localhost:8080/actuator/beans.

TODO-04 Expose all HTTP actuator endpoints.

- 1. Modify the property again to expose *all* actuator endpoints. Once the application restarts, visit the following endpoints:
 - http://localhost:8080/actuator/beans
 - http://localhost:8080/actuator/health
 - http://localhost:8080/actuator/info
 - http://localhost:8080/actuator/mappings
 - http://localhost:8080/actuator/loggers
 - http://localhost:8080/actuator/metrics/jvm.memory.max
- 2. Use tags with metrics endpoint
 - http://localhost:8080/actuator/metrics/http.server.requests
 - http://localhost:8080/actuator/metrics/http.server.requests?tag=method:GET
 - http://localhost:8080/actuator/metrics/http.server.requests?tag=uri:/actuator/beans
- 3. Access some URL endpoints that are not existent and get metrics on them.
 - http://localhost:8080/notexistent
 - http://localhost:8080/actuator/metrics/http.server.requests?tag=status:404

Change Logging Level

TODO-05 Change log level via ./actuator/loggers endpoint

- 1. Display logging level of account.web package
 - http://localhost:8080/actuator/loggers/accounts.web

Observe that the effectiveLevel is currently set to DEBUG

```
{
   "configuredLevel": null,
   "effectiveLevel": "DEBUG"
}
```

2. Add a log statement as shown below to the accountSummary() method of the controller

```
@GetMapping(value = "/accounts")
public @ResponseBody List<Account> accountSummary() {
   logger.debug("Logging message within accountSummary()"); // add this line
   return accountManager.getAllAccounts();
}
```

3. Once the application restarts, access /accounts url and verify that the log message gets displayed

4. Change the logging level to INFO using either curl or Httpie or Postman

```
curl -i -XPOST -H"Content-Type: application/json" localhost:8080/actuator/loggers/accounts.web -d'{"configuredLevel"
```

```
http post localhost:8080/actuator/loggers/accounts.web configuredLevel=INFO
```

5. Verify that the effectiveLevel of the accounts.web package is now changed to INFO

```
{
  "configuredLevel": "INFO",
  "effectiveLevel": "INFO"
}
```

6. Access /accounts url again and verify that the log message no longer gets displayed

Publish Build Information

Spring Boot Actuator's info endpoint publishes information about your application specified in the META-INF/build-info.properties file.

The META-INF/build-info.properties can be created by the Spring Boot Maven or Gradle Plugin during build

TODO-06 Add Maven goal or Gradle task

1. If you are using Maven, add build-info goal

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <executions>
        <execution>
            <goals>
                  <goal>build-info</goal>
                  </goals>
                  <execution>
                  </execution>
                  </execution>
                  </execution>
                  </executions>
                  </plugin>
```

if you are using Gradle, add BuildInfo task

```
springBoot {
  buildInfo()
}
```

2. Rebuild the application preferably at the command line

```
./mvnw -pl 00-rewards-common -pl 01-rewards-db -pl 44-actuator clean install
```

```
./gradlew 44-actuator:clean 44-actuator:build
```

- 3. Verify that target/classes/META-INF/build-info.properties (for Maven) or build/resources/main/META-INF/build-info.properties (for Maven) or build/resources/main/META-INF/build-info.properties/main/META-INF/build-info.properties/main/META-INF/build-info.properties/main/META-INF/build-info.properties/main/META-INF/build-info.properties/main/META-INF/build-info.properties/main/META-INF/build-info.properties/main/META-INF/build-info.properties/main/META-INF/build-info.properties/main/META-INF/build-info.properties/main/META-INF/build-info.properties/main/META-INF/build-info.properties/main/META-INF/build-info.properties/main/META-IN
- 4. Visit info endpoint and verify that the build info gets displayed

1. Expose the Java runtime information. The Java info contributor is disabled by default, enable it by adding the following property in the application.properties

```
management.info.java.enabled=true
```

2. Add some custom properties to the application.properties

```
management.info.env.enabled=true
info.restaurant.location=New York
info.restaurant.discountPercentage=10
```

- 3. Rebuild the application
- 4. Visit info endpoint and verify that additional properties are displayed

Define Custom Metrics - Counter

By default, Actuator exposes a number of useful but rather generic metrics. Often tracking metrics that are specific to your application's domain provides into operational, business, or other concerns.

Let's add a custom metric specific to our application that counts the number of times the account finder method is used.

TODO-08: Add a counter with a tag

- 1. In the constructor of AccountController, add an instance of MeterRegistry as a second parameter.
- 2. Using the MeterRegistry, create a Counter called account.fetch with a tag of type/fromCode key/value pair by calling meterRegistry.counter("account.fetch", "type", "fromCode");
- 3. Store the counter as a new data-member.

TODO-09: Use the counter

1. In the accountDetails() method, call increment() on the Counter.

TODO-10: Run the test

1. Verify that the tests in AccountControllerTests pass. Particularly the testHandleDetailsRequest(), which checks that the counter i working correctly.

TODO-11: Verify the result

- 1. Once the application restarts, visit the (http://localhost:8080/actuator/metrics endpoint again. You should now see account.fetch listed.
- 2. Visit the http://localhost:8080/actuator/metrics/account.fetch to view the data for your counter. This should display 0 since no accounts have been fer yet.
- 3. Fetch an account by visiting http://localhost:8080/accounts/1.
- 4. Visit the (http://localhost:8080/actuator/metrics/account.fetch again, verify that the counter increase with each account fetch.

Try restarting your application. What happens to the counter?

Define Custom Metrics - Timer

TODO-12: Add timer using @Timed annotation

1. Add the following annotation to the accountSummary(..) method.

```
@Timed(value="account.timer", extraTags = {"source", "accountSummary"})
```

2. Add the following annotation to the accountDetails(...) method.

```
@Timed(value="account.timer", extraTags = {"source", "accountDetails"})
```

TODO-13: Verify the result

- 1. Once the application restarts, visit the localhost:8080/accounts/1 and localhost:8080/accounts for a few times
- 2. Visit the (http://localhost:8080/actuator/metrics/account.timer and verify the timer metric

Get Detail Health Checks

TODO-14: Add more details to the health check

1. Visit the http://localhost:8080/actuator/health endpoint.

By default there is very little info displayed at this endpoint.

- 2. In your application.properties file, enable more detailed health info by setting the management.endpoint.health.show-details to always.
- 3. Restart the application and refresh http://localhost:8080/actuator/health to see more health details.

Create Custom Health Checks

You can extend the default health checks so that your application reports whether it is down or up based on custom criteria or domain logic.

In this case, we will determine the health of the application based on whether there are any restaurants in the database. If there are no restaurants, then health status of the application will be considered DOWN.

NOTE: Stop the ActuatorApplication to avoid it constantly restarting whilst we add new classes.

TODO-15a and TODO-15b: Setup a test

- 1. Navigate to the src/test/java directory. Inside the accounts.web package, there is a class called RestaurantHealthCheckTest. It is mostly write vou.
- 2. Modify the code to use the RestaurantHealthCheck.health() method in each test. The code will not compile yet we have not written the code v testing.

TODO-16a: Implement RestaurantHealthCheck

- 1. Navigate to the RestaurantHealthCheck class under the accounts.web package.
- 2. Modify the class to implements the HealthIndicator interface.
- 3. Implement the missing health() method with the following logic:
 - If there are one or more restaurants in the database, return Health.up().build().
 - Otherwise return Health.down().build().
 - You will need to pass the RestaurantRepository into this object via constructor injection.

TODO-16b: Test custom health indicator

- 1. Create an instance of RestaurantHealthCheck class in the setup method in the RestaurantHealthCheckTest
- 2. Remove @Disabled annotation from both tests
- 3. Run the tests and verify they pass.

- 1. Restart the application.
- 2. Refresh the health endpoint. You should see that your application is DOWN because there are no Restaurants in the database.

TODO-18/19: Repeat with restaurants in the database

- 1. To populate your database with a Restaurant, change the application.properties to set spring.sql.init.data-locations property with classpath:/data-with-restaurants.sql.
- 2. Once the server restarts refresh the health endpoint. You should now see that your application health reports UP.
- 3. Verify that all tests in the actuator project pass.

Organize Health Indicators Into Groups

TODO-20: Organize Health Indicators into groups

From Spring Boot 2.2, health indicators can be organized into groups using the format below.

```
management.endpoint.health.group.<group-name>.include=<list of health indicators>
```

Each group can be individually configured.

- 1. Create 3 groups: system, web, and application as following:
 - The system group includes diskSpace and db health indicators
 - The web group includes ping health indicator
 - The application group includes restaurantHealthCheck health indicator
- 2. For system and application groups, configure show-details with always
- 3. Remove management.endpoint.health.show-details=always since it is no longer needed
- 4. Restart the application and access the health endpoint per each group
 - http://localhost:8080/actuator/health/system
 - http://localhost:8080/actuator/health/web
 - http://localhost:8080/actuator/health/application

Secure Actuator Endpoints

TODO-21: Add Spring Boot Security starter to the pom.xml or build.gradle file

TODO-22: Add security configuration

1. The skeleton code of the security configuration class, ActuatorSecurityConfiguration is already provided. Uncomment the code there is no compile errors.

TODO-23: Add security configuration to ActuatorSecurityConfiguration class

1. Add code to provide access control to actuator endpoints

- 1. Take some time to understand what each test is for
- 2. Remove @Disabled annotation from each test and run it
- 3. Make sure all tests pass

Summary

In this lab you have configured Spring Boot Actuator. You have also leveraged Spring Boot Actuator to give detailed information about application metrics customized health stats.

Congratulations, you are done with the lab. If this were a production application, one logical next step would be to integrate Actuator with an external mor system.

Optional Exercises

Do the following exercises if you have an extra time.

TODO-25 (Optional): Experiment with HealthIndicator

- 1. Change spring.sql.init.data-locations property in the application.properties file back to use classpath:data-no-restaurants.sql.
- 2. Include the restaurant count as extra detail in the health endpoint. Have a look at the Health class to see how this might work.
- 3. Instead of returning DOWN when there are no restaurants, use a custom status called NO_RESTAURANTS. You will have to create Status object.
- 4. When there are no restaurants in the DB, what top-level status is returned for the application health group?

Fix this issue by changing management.endpoint.health.group.application.status.order property in the "application.properties" file so that NO_RESTAURANTS gets displayed as top-level status for the application health group.

5. Restart the application and verify the result.

TODO-26 (Optional): Use AOP for counting logic

If you are short on time, skip this step.

In general, mixing up different concerns (controller logic and counter logic in this example code) is not considered a good practice: it violates *Single Responsibility Principle*. Instead, usage of AOP provides cleaner code.

- 1. Add spring-boot-starter-aop starter to the pom.xml or the build.gradle
- 2. Create an aspect, through which account.fetch counter, which has a tag of type/fromAspect key/value pair, gets incremented every time accountSummary method of the AccountController class is invoked
- 3. Access /accounts several times and verify the metrics of /actuator/metrics/account.fetch?tag=type:fromAspect

TODO-27 (Optional) : Access Actuator endpoints using JMX

If you are short on time, skip this step.

• JMX is disabled by default from Spring Boot 2.2. Enable it by adding the following line to the application.properties file

spring.jmx.enabled=true

- Restart the application
- Open a terminal window and run jconsole. Accept the insecure connection if prompted

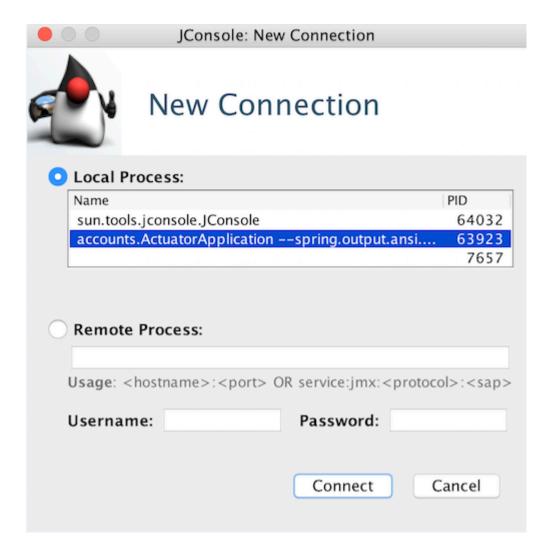


Figure 1: Starting JConsole

• Select the MBeans tab, find the org.springframework.boot folder, then open the Endpoint sub-folder. Note that all actuator endpoints ARE expose JMX

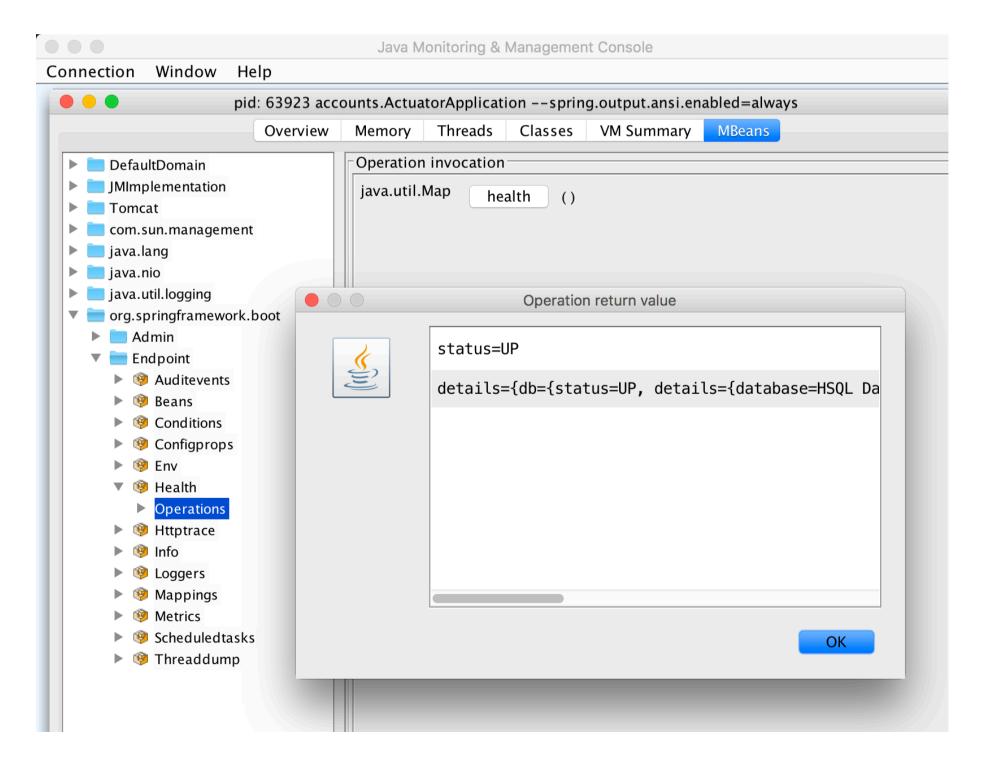


Figure 2: Spring Boot endpoints in JConsole

Summary Instructors

In this lesson you exercised the features of Actuator.

☐ Give Feedback

Help us improve by sharing your thoughts.

Give Feedback



Guides | Courses | Learning Path | Community | Instructors | About | Give Feedback

Contact | FAQs | Terms | Privacy | Your California Privacy Rights

Unlock your full potential with Spring courses designed by experts.

Copyright © 2005-2024 Broadcom. All Rights Reserved. The term "Broadcom" refers to Broadcom Inc. and/or its subsidiaries.