CST-339 Activity 8 Guide

Contents

Optional Enterprise Features

Overview

Goal and Directions:

The following are optional (not graded) activities that you can complete or reference as needed to add enterprise class features that will make your application enterprise-ready and ready to be supported in the Cloud. The enterprise features include adding application logging, REST API data validation, global exception handling for both Web and Rest Controllers, method tracing for the application, encrypting passwords, using profiles for specifying various configuration parameters for different environments, an approach for keeping your REST API documentation always up to date and in sync with your code, and finally, deploying your application to Docker Containers.

Make a copy of the *topic5.1* project by right-clicking on the project, selecting the Copy menu option, then right-clicking in the Workspace, and then selecting the Paste menu option. Name the project *topic8*. Rename the Spring application file from *Topic51Application.java* to *Topic8Application.java*. Fix up the *artifactId* and *name* tags in the POM file (pom.xml) so the value is set to *topic8*. Run the project to ensure that everything is working properly.

1. **DevOps Logging Instrumentation:** In order to debug your application in a production environment or in the Cloud, it is often necessary to add log statements in your code, as well as log errors and exceptions. This can be done with logging to your application. To add logging to your application, complete the following steps:
    a. In Spring Boot, the Logback Logger is enabled by default, so no additional dependencies need to be added to your application. For this activity, logging will be added to the *OrdersRestService* class, but the same steps can be used for any class in your application. Add a class-scoped member variable named *logger* of type *Logger* that gets an instance of a *Logger* from the *LoggerFactory.getLogger()* method. All classes should resolve to the *org.slf4j* package. Call the desired *info(), warn(), trace(), debug(), or error()* methods on the *logger* variable as desired in your application code.

```
private static final Logger logger = LoggerFactory.getLogger(OrdersRestService.class);
```

    b. Customize the Logback Logger by adding a file named *logback-spring.xml* file in the *src/main/resources* directory. An example configuration file is shown below

that configures a log file and GCU console logger that, as shown, filters on the trace level (change this as desired).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <property name="LOGS" value="./logs" />

    <appender name="Console" class="ch.qos.logback.core.ConsoleAppender">
        <layout class="ch.qos.logback.classic.PatternLayout">
            <Pattern>
                %d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n
            </Pattern>
        </layout>
    </appender>

     <appender name="GcuConsole" class="ch.qos.logback.core.ConsoleAppender">
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>TRACE</level>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
        </filter>
        <layout class="ch.qos.logback.classic.PatternLayout">
            <Pattern>
                %d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n
            </Pattern>
        </layout>
    </appender>

    <appender name="RollingFile" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${LOGS}/spring-boot-logger.log</file>
        <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <Pattern>%d %p %C{1.} [%t] %m%n</Pattern>
        </encoder>

        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- rollover daily and when the file reaches 10 MegaBytes -->
            <fileNamePattern>${LOGS}/archived/spring-boot-logger-%d{yyyy-MM-dd}.%i.log</fileNamePattern>
            <timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
                <maxFileSize>10MB</maxFileSize>
            </timeBasedFileNamingAndTriggeringPolicy>
        </rollingPolicy>
    </appender>

    <!-- LOG everything at INFO level -->
    <root level="info">
        <appender-ref ref="RollingFile" />
        <appender-ref ref="Console" />
    </root>

    <!-- LOG "com.gcu" at TRACE level -->
    <logger name="com.gcu" level="trace" additivity="false">
        <appender-ref ref="RollingFile" />
        <appender-ref ref="GcuConsole" />
    </logger>
</configuration>
```

2. **DevOps Logging Instrumentation:** In order to debug your application in a production environment or in the Cloud, it is often necessary to determine what a person was doing in the application and what flows in the application were being executed. This can be

done with method tracing. To add Method Tracing to an application, complete the following steps:

a. Add the Spring AOP dependency to the POM file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

b. In the *com.gcu.util* package, create a *Tracer* utility class that extends from the Spring *AbstractMonitoringInterceptor* class. Create a default constructor and non-default constructor that takes a boolean method argument that will utilize the default dynamic Spring logger (this could be changed if you want to use your own static logger class). Override and implement the *invokeUnderTrace* method that will provide the custom logic to be used during method tracing. An example below shows tracing logic that will log and measure the performance of each method. It should also be noted that this implementation is using the trace log level, which was enabled by default in the GCU Console Logger in the previous activity. This logic can be changed as desired.

```java
public class Tracer extends AbstractMonitoringInterceptor
{
    private static final long serialVersionUID = -5378974652504403928L;

    public Tracer()
    {
    }

    // If true will use the dynamic Spring logger
    public Tracer(boolean useDynamicLogger)
    {
        setUseDynamicLogger(useDynamicLogger);
    }

    @Override
    protected Object invokeUnderTrace(MethodInvocation invocation, Log log) throws Throwable
    {
        // Before Method Invocation display a start method trace log statement
        String name = createInvocationTraceName(invocation);
        long start = System.currentTimeMillis();
        log.trace("GCU Method " + name + " execution started at:" + new Date());
        try
        {
            // Invoke the method
            return invocation.proceed();
        }
        finally
        {
            // After Method Invocation display an end method trace log statement
            long end = System.currentTimeMillis();
            long time = end - start;
            log.trace("GCU Method " + name + " execution lasted:" + time + " ms");
            log.trace("GCU Method " + name + " execution ended at:" + new Date());
            if (time > 10)
                log.warn("GCU Method execution longer than 10 ms!");
        }
    }
}
```

c. In the *com.gcu.util* package create an *AopConfiguration* utility class that will be used to wire up the Tracing Interceptor to the desired points configured in this class. The class should be marked with the *@Configuration* and *@EnableAspectJAutoProxy* annotations and implement the following configuration logic.

```java
@Configuration
@EnableAspectJAutoProxy
public class AopConfiguration
{
    // Setup Pointcuts to the Controllers, Rest Controllers, Business Services, and Data Services
    @Pointcut("execution(* com.gcu..controller..*(..)) || execution(* com.gcu..business..*(..)) || execution(* com.gcu..data..*(..))")
    public void monitor()
    {
    }

    // Get an instance of the Tracer that will be used in the Aspect
    @Bean
    public Tracer tracer()
    {
        return new Tracer(true);
    }

    // Setup the Aspect with the Tracer and reference to the monitor() Pointcut
    @Bean
    public Advisor performanceMonitorAdvisor()
    {
        AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();
        pointcut.setExpression("com.gcu.util.AopConfiguration.monitor()");
        return new DefaultPointcutAdvisor(pointcut, tracer());
    }
}
```

d. Run the web application and REST API's to exercise all of the functionality. There should log trace statements that start with the prefix of "GCU Method."

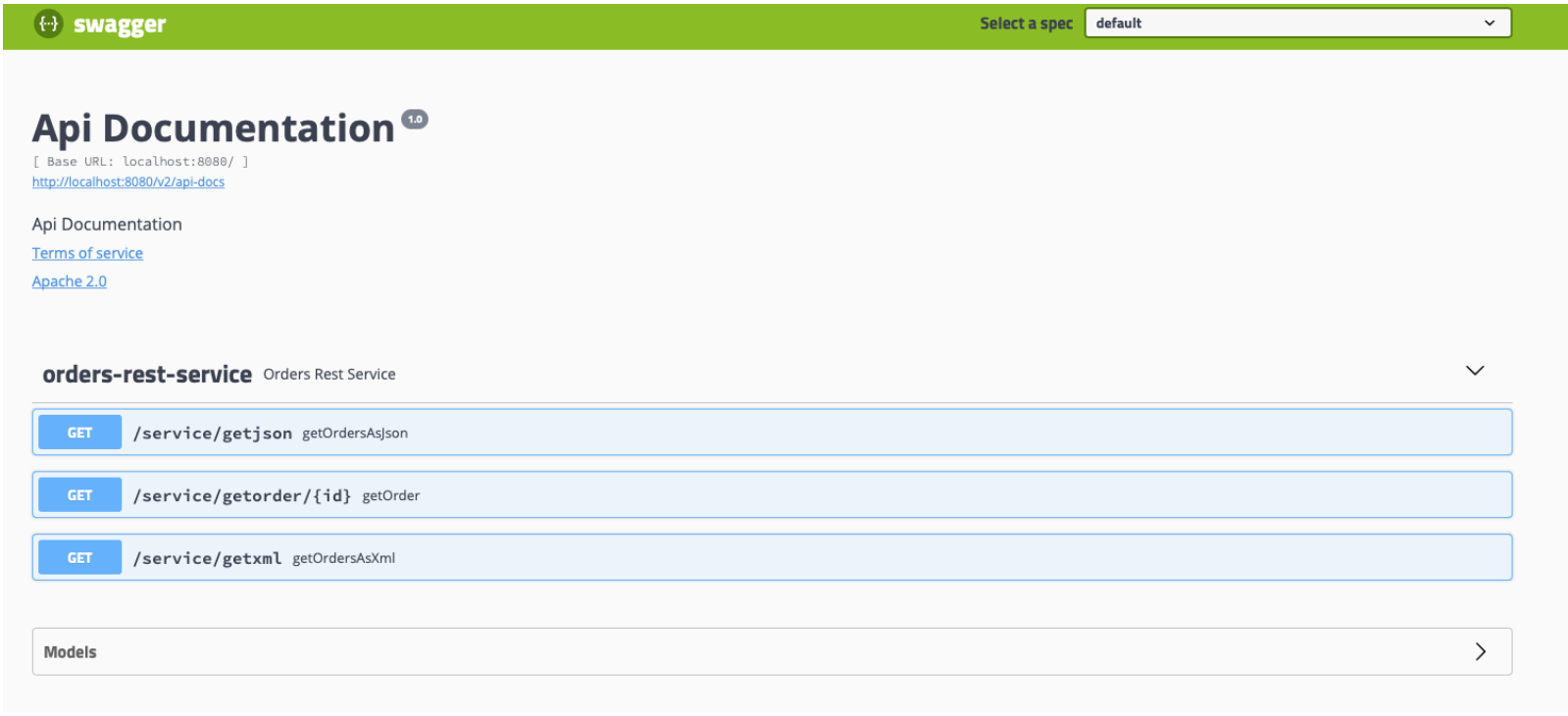
3. **REST API's:** It is always important to keep your REST API documentation up-to-date and in sync with your code, either for other developers or the clients of your API's. To automatically generate Swagger REST API documentation that is always up-to-date and in sync with your code, complete the following steps:
   a. Add the Springfox Swagger2 dependencies to the POM file.

```xml
<!--  For Springfox Swagger2 -->
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

   b. Mark the *Topic8Application* class with the *@EnableSwagger2* annotation with the Docket bean declaration with a default implementation. The path selector can be customized as needed for your application based on the package specification and where your REST API's exist in your project.
   c.

```
@Bean
public Docket api()
{
    return new Docket(DocumentationType.SWAGGER_2)
      .select()
      .apis(RequestHandlerSelectors.any())
      .paths(PathSelectors.ant("/service/**"))
      .build();
}
```

d. Access the REST API documentation application in your browser by navigating to the *http://localhost:8080/swagger-ui.html* URL.



4. There are lots of customization options with many annotations that can be added to the REST API classes (using the *@Api()* annotation) and application object model classes (using the *@ApiModel()* annotation). See the Springfox home page here for additional information and specifically chapter 6 for how to further customize the metadata and user interface of the documentation.
5. Much like you would validate data entered in a form by a user in a web application, it is just as vital to validate the data that is sent to your REST API's. To add data validation to your REST API's complete the following steps:
    a. Ensure that you have the *spring-boot-starter-validation* dependency in your POM file.
    b. For this activity, data validation will be added to the *OrdersRestService* class, but the same steps can be used for any REST API class in your application. Any error

during path or request validation in Spring results by default in an HTTP 500 response code being returned to the client.

c. Add the *@Validated* annotation to the class level of the *OrdersRestService* class.
d. Add the following data validation rules to the method argument to validate the specified Order ID: *@NotBlank @Size(max = 24)*.
    i. It should be noted that you can also validate HTTP Parameters by simply adding the data validation rules to the method arguments.
    ii. It should be noted that you can also validate a Model by simply adding the data validation rules to the Model properties.

6. **Exception Handling:** Spring adds to ability to easily add common exception handlers to all your MVC Web Controllers. This allows you to have a single place in your code where common code for exception handling avoiding the need to repeatedly add the same code to many controller classes. To add a Global Exception Handler for all controller classes, complete the following steps:
    a. Create a view named *exception* that displays a passed Error Message attribute named *errorMessage*.
    b. In the *com.gcu.controller* package, create a new class named *ExceptionControllerAdvice* that is marked with the *@ControllerAdvice* annotation as the class level. The class should implement a public method named *ModelAndView handleException(Exception ex)* that is marked with the *@ExceptionHandler(Exception.class)* annotation and creates and returns a Model and View to the *exception* view. It should be noted that you can create additional methods marked with the desired *@ExceptionHandler* annotation with different exception classes to customize the handling of different exceptions as desired.

```
@ControllerAdvice
public class ExceptionControllerAdvice
{
    @ExceptionHandler(Exception.class)
    public ModelAndView handleException(Exception ex)
    {
        // Create a Model and View, populate with the Exception information, and display a common Error Page
        ModelAndView model = new ModelAndView();
        model.addObject("title", "Error Page");
        model.addObject("errorMessage", "An Exception was not handled in the application: " + ex.getMessage());
        model.setViewName("exception");
        return model;
    }
}
```

    c. You can test this handler out by throwing a Runtime or Exception in one of the Controller methods.

```
// Get some Orders
List<OrderModel> orders = service.getOrders();
if(orders != null)
    throw new RuntimeException("Something bad happened");
```

7. **Exception Handling:** Spring adds to ability to easily add common exception handlers to all your REST Controllers. This allows you to have a single place in your code for common code for exception handling, avoiding the need to repeatedly add the same code to many controller classes. To add a Global Exception Handler for all Rest Controller classes, complete the following steps:
    a. In the *com.gcu.business* package, create a new class named *RestExceptionControllerAdvice* that is marked with the *@RestControllerAdvice* annotation as the class level. The class should implement a public method named *ResponseEntity<?> handleException(Exception ex)* that is marked with the *@ExceptionHandler(Exception.class)* annotation and creates and returns a *ResponseEntity* set with a desired data payload and an *HttpStatus.INTERNAL_SERVER_ERROR* status code. It should be noted that you can create additional methods marked with the desired *@ExceptionHandler* annotation with different exception classes to customize the handling of different exceptions as desired.
    b. You can test this handler out by throwing a Runtime or Exception in one of the Rest Controller methods.

```
OrderList list = new OrderList();
if(list != null)
    throw new RuntimeException("Something bad happened");
```

8. **Security:** You should never store passwords in clear text, and that includes in your code and source control system. You can encrypt your database credentials that are configured in your application property file by using the following steps:
    a. Add the Jasypt library to your Maven project.

```
<!-- For Jasypt to Encrypt Passwords-->
<dependency>
    <groupId>com.github.ulisesbocchio</groupId>
    <artifactId>jasypt-spring-boot-starter</artifactId>
<version>2.1.0</version>
```

    b. Encrypt your credentials by going here.
    c. Update your application property file with the encrypted credentials from Step b and enclose the encrypted text in ENC() as shown below.

```
spring.datasource.username=ENC(2ZanVNv7l8UWFWDgnAcSIjdvxkyAPUAd+8CcYzBQwHk=)
spring.datasource.password=ENC(tcZg2HhX9r3bXXcqsRMz2pfa0C+JWc3RihXG+ZVIr8E=)
```

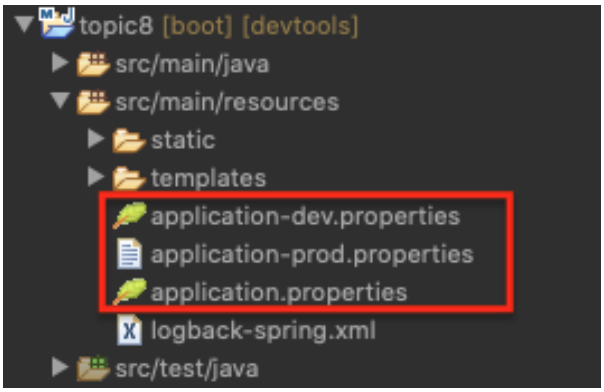
    d. Enable property encryption in your application by adding the following annotation to the Spring Application class:

```
@EnableEncryptableProperties
```

    e. Set an environment variable named JASYPT_ENCRYPTOR_PASSWORD to your secret key set in Step b. FOR TESTING ONLY, you can set the property

jasypt.encryptor.password to the value of your secret key in your application property file.

9. **Configuration:** It is often necessary to have different configuration setting for various environments, such as Development, QA, and Production, as well as for different cloud platforms. This capability is supported in Spring through the use of Spring Profiles. It should be also noted that Maven also has a similar capability, but in Maven, this works at a different level and at build time whereas Spring Profiles works at runtime. Spring Profiles are supported in a number of ways, but one way is through property files (and also YAML files). To use profile specific property files in your application, complete the following steps:
    a. Create the desired number of application property files for each profile you need to support by following a naming convention of *application-[PROFILE_NAME].properties*. Set the appropriate property values in each application properties file for each environment you need to support.



    b. The active profile can be set in a number of ways:
        i. spring.profiles.active=*[PROFILE_NAME]* in *application.property* file.
        ii. -Dspring.profiles.active=*[PROFILE_NAME]* in a JVM system parameter.
        iii. export spring.profiles.active=*[PROFILE_NAME]* in an Operating System Environment variable.
        iv. Key of spring.profiles.active and a value of *[PROFILE_NAME]* in a cloud environment (or secret) variable.

```
# Set the active Spring Profile
spring.profiles.active=dev
```

10. **Docker Containers:** One of the advantages of using Spring Boot and its capability to package your application with the embedded runtime, such as the embedded Tomcat Server, is the ability to easily build and deploy monolith and microservice-based applications into Docker Containers. The following outlines how this can be done.
    a. Google provides a Maven Build plugin that enables you at build time of your code to also build a Docker Image and push this image to a Docker Repository. The Maven Build plugin is called Jib. One of the advantages of Jib is that the

b. Dockerfile is built on the fly through XML configuration with a number of widely used Docker Repositories supported. Documentation for Jib can be found here. An example Jib build script is shown below:

```
<plugin>
    <!-- Run this by adding the following goal: jib:dockerBuild -->
    <!-- TO GET THIS TO WORK YOU MUST RUN SPRING TOOL SUITE AS SUDO SO IT CAN ACCESS DOCKER DAEMON (sudo open -a SpringToolSuite4.app)!!!! -->
    <!-- Run the application with the following docker command: docker run -p8090:8080 -it markreha/playspringboot:1.0.0 and go to http://localhost:8090/ -->
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>jib-maven-plugin</artifactId>
    <version>2.4.0</version>
    <configuration>
        <from>
            <image>openjdk:8-jre-alpine</image>
        </from>
        <to>
            <image>markreha/playspringboot:1.0.0</image>
        </to>
        <container>
            <ports>
                <port>8080</port>
            </ports>
        </container>
    </configuration>
</plugin>
```

c. Docker Containers are also supported with the free Heroku cloud platform. Deploying a Docker Image to Heroku is done via their Command Line Interface (CLI). An example application is shown below.
   i. Create an application in Heroku either using the CLI or the web interface. The application shown in this example is named *playspringboot2-mkr*.
   ii. Build a Docker Image either through Jib or via a Dockerfile. The image should be built from the *openjdk:8-jdk-alpine* base image, contain your application in a file named *app.jar*, expose port *8080*, and also set the *server.port* variable to *$PORT* in the Java startup command. The Docker Image shown in this example is named *markreha/playspringboot2:1.0.0*.

```
FROM openjdk:8-jdk-alpine
COPY target/playapplication.jar app.jar
EXPOSE 8080
CMD [ "sh", "-c", "java -jar $JAVA_OPTS -Xmx300m -Xss512k -Dserver.port=$PORT /app.jar" ]
```

```
docker build -t markreha/playspringboot2:1.0.0 .
```

   iii. Run the Docker Image to make sure it works without error.

```
docker run -p8090:8080 -it markreha/playspringboot2:1.0.0
```

   iv. Using the Heroku CLI, log into Heroku, tag and push the built Docker Image to the Heroku Docker Repository, release the Docker Image to your application, and open your application in Heroku.

```
heroku container:login
docker tag markreha/playspringboot2:1.0.0 registry.heroku.com/playspringboot2-mkr/web
docker push registry.heroku.com/playspringboot2-mkr/web
heroku container:release web -a playspringboot2-mkr
heroku open -a playspringboot2-mkr
```

11. **Microservices Research:** Topic 7 provided an introduction to microservices in Spring Boot. However, there are a number of questions that must be addressed when building the architecture for an enterprise class, microservices-based application. The following are a few questions that should be research and answered:
    a. What are the impacts on Logging, and why is a log aggregator a requirement in a microservices architecture? How will the log files be correlated with each other?
    b. What are the impacts on the network when moving to a microservices architecture?
    c. What are the impacts on ACID database transactions when moving to a distributed microservices architecture?
    d. What are the impacts of causing potential circular references between services, and how could a circuit breaker pattern be used when moving to a microservices architecture?

e. What are the impacts of splitting a production database into separate domains and databases when moving to a microservices architecture? Are there short-term approaches you can take to avoid design and data migration issues?
f. What are the impacts on existing CI/CD pipelines when moving to a microservices architecture?
g. Why is a REST API versioning strategy essential for a successful microservices architecture?
h. Why is an accurate and up-to-date REST API documentation strategy essential for a successful microservices architecture?

Final Activity Submission

1. This activity is optional. Therefore, nothing is required to be submitted.