5COSC022W.2 Client-Server Architectures

Tutorial Week 10: RESTful web services with JAX-RS

INTRODUCTION

In this tutorial you will implement ExceptionMappers. An ExceptionMapper in JAX-RS is an interface that allows you to customize the response sent to the client when an exception occurs during the processing of a request. By implementing the ExceptionMapper, you can define how the application should respond with a specific HTTP status code and response body when that exception is thrown. This helps in creating a consistent and user-friendly API by providing meaningful error messages and appropriate status codes. For example, you could map a `NotFoundException` to a 404 status code, while a `WebApplicationException` could return a 500 status code depending on the nature of the error. ExceptionMappers enhance error handling in your RESTful services and make them easier to debug and maintain.

REQUIREMENTS

- Basic knowledge of Java
- NetBeans 18 or above
- Apache Tomcat server

EXERCISE 1

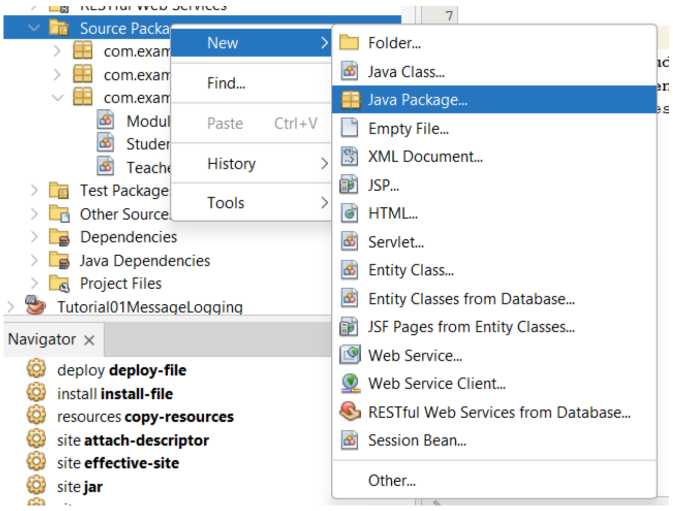
In this exercise, you will download a ZIP file that includes the project you implemented during the last tutorial session.

STEP 1: IMPORT THE PROJECT

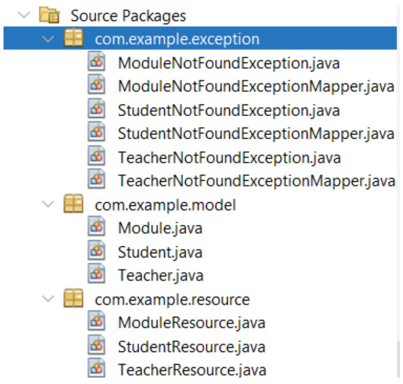
- You need to download the ZIP file called Tutorial_Week09_NO_DAO.zip from the blackboard under Week 10.
- Then, you need to import it to NetBeans.

STEP 2: COM.EXAMPLE.EXCEPTION PACKAGE

- Under the source packages, please create another package, name it as com.example.exception.



- After creating the package, please follow these steps to implement custom exceptions and exception mapper classes. The project structure should look like this:



```
Source Packages
  com.example.exception
    ModuleNotFoundException.java
    ModuleNotFoundExceptionMapper.java
    StudentNotFoundException.java
    StudentNotFoundExceptionMapper.java
    TeacherNotFoundException.java
    TeacherNotFoundExceptionMapper.java
  com.example.model
    Module.java
    Student.java
    Teacher.java
  com.example.resource
    ModuleResource.java
    StudentResource.java
    TeacherResource.java
```

STEP 3: JAVA CLASSES INSIDE COM.EXAMPLE.EXCEPTION PACKAGE

StudentNotFoundException Class Instructions:

1. **Class Definition:**
   - Create a `public` class named `StudentNotFoundException` that `extends` `RuntimeException`. This makes it an unchecked exception.
2. **Constructor:**
   - Create a `public` constructor that takes a `String message` as a parameter.
   - Inside the constructor, call the superclass constructor (the constructor of `RuntimeException`) using `super(message);`. This passes the message to the `RuntimeException` class, which stores it so it can be retrieved later (e.g., using `getMessage()`).

StudentNotFoundExceptionMapper Class Instructions

1. **Class Definition:**
   - Create a `public` class named `StudentNotFoundExceptionMapper` that `implements ExceptionMapper<StudentNotFoundException>`. This tells JAX-RS that this class will handle `StudentNotFoundException` instances.
2. **Imports:**
   - Import the following classes:
     - `javax.ws.rs.core.Response`
     - `javax.ws.rs.ext.ExceptionMapper`
     - `javax.ws.rs.ext.Provider`
     - `org.slf4j.Logger`
     - `org.slf4j.LoggerFactory`
     - `javax.ws.rs.core.MediaType`
3. **`@Provider` Annotation:**
   - Add the `@Provider` annotation *above* the class definition. This annotation tells JAX-RS to register this class as a component (specifically, an exception mapper). JAX-RS will automatically discover and use it.
4. **Logger**
   - Declare a `private static final Logger` named `logger`. Initialize it using `LoggerFactory.getLogger(StudentNotFoundExceptionMapper.class)`.
5. **`toResponse` Method:**
   - Override the `toResponse` method from the `ExceptionMapper` interface. This method takes a `StudentNotFoundException` object as a parameter and returns a `Response` object. Add the `@Override` annotation above the method definition.
   - **Log Error:**
   - Inside the `toResponse` method, log the exception using the `logger`. Use the `error` level and include the exception message. Include the `exception` object itself in the logging call to get a full stack trace in the logs: `logger.error("Student not found: {}", exception.getMessage(), exception);`
   - **Build Response:**
     - Create a `Response` object using the builder pattern:
       - Start with `Response.status(Response.Status.NOT_FOUND)` to set the HTTP status code to 404 (Not Found).

        - Chain `.entity(exception.getMessage())`: Set the response body to the *message* from the `StudentNotFoundException`.
        - Chain `.type(MediaType.TEXT_PLAIN)`: Set the `Content-Type` header to `text/plain`. This indicates that the response body is plain text.
        - Call `.build()` to create the final `Response` object.
    - `return` the created `Response` object.

ModuleNotFoundExceptionMapper and TeacherNotFoundExceptionMapper Class Instructions:

Repeat the same steps that you did for Student Exception Mapper class. Just change the names to be related to Module and Teacher.

STEP 4: POM FILE

ADDING DEPENDENCIES AND PLUGINS:

Please check to make sure the following dependencies and plugins are included in **pom.xml**.

```
<dependencies>
        <dependency>
            <groupId>org.glassfish.jersey.inject</groupId>
            <artifactId>jersey-hk2</artifactId>
            <version>2.32</version>
        </dependency>
        <dependency>
            <groupId>org.glassfish.jersey.containers</groupId>
            <artifactId>jersey-container-servlet</artifactId>
            <version>2.32</version>
        </dependency>
        <dependency>
            <groupId>org.glassfish.jersey.media</groupId>
            <artifactId>jersey-media-json-jackson</artifactId>
            <version>2.32</version> <!-- Adjust version as needed -->
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>8</source>
                    <target>8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.2.2</version>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

CONFIGURING WEB.XML

In the current web.xml file we have only com.example.resource package under <init-param>. To use exceptions, you need to also add com.example.exception which is highlighted in green.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">
    <servlet>
        <servlet-name>StudentApplication</servlet-name>
        <servlet-class>
org.glassfish.jersey.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>jersey.config.server.provider.packages</param-name>
            <param-value>com.example.resource, com.example.exception</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>StudentApplication</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

- Please make sure you have saved both **web.xml** and **pom.xml**

STUDENT RESOURCE CLASS

`StudentResource` Class Instructions (with Exception Handling)

1. **Class Definition:**
    - The class should already be defined as `public class StudentResource`.
2. **Imports:**
    - Make sure these imports are present (add them if they are missing):
        - `com.example.exception.StudentNotFoundException`

        - `com.example.model.Module`
        - `com.example.model.Student`
        - `java.util.ArrayList`
        - `java.util.List`
        - `javax.ws.rs.*`
        - `javax.ws.rs.core.MediaType`
        - `org.slf4j.Logger`
        - `org.slf4j.LoggerFactory`
3. **Class-Level Annotations:**
    - `@Path("/students")` should already be present.
4. **Fields:**
    - `private static final Logger logger = LoggerFactory.getLogger(StudentResource.class);`
    - `private static List<Student> students = new ArrayList<>();`
    - `private static int nextId = 3;`
5. **Static block**
    - No changes required in the static block. Please note that in this exercise, all data operations are handled inside the resource class and there is no DAO classes. You may either use this approach in your coursework or DAO implementation.

6. **`getAllStudents` Method:** (add logging only)
    - The method should already exist and:
        - Be annotated with `@GET` and `@Produces(MediaType.APPLICATION_JSON)`.
        - Return the `students` list.
        - Log information message: "GET request for all students"
7. **`getStudentById` Method:** (Modify to use `orElseThrow`)
    - The method should already exist with these annotations: `@GET`, `@Path("/{studentId}")`, `@Produces(MediaType.APPLICATION_JSON)`, and `@PathParam("studentId")`.
    - **Change the Implementation:**
        - Log information message: "GET request for student with ID: {}" with studentId.
        - Use the Java Stream API to find the student. This is the *most important* change for exception handling. The method body should look like this in terms of instructions:
            - Start with `students.stream()`.
            - Chain `.filter(student -> student.getId() == studentId)` to filter the list.
            - Chain `.findFirst()` to get an `Optional<Student>`.

        - Chain `.orElseThrow(() -> new StudentNotFoundException("Student with ID " + studentId + " not found"));`. This is key: If a student is found, `.orElseThrow` does nothing and the `Student` object is returned. If *no* student is found, `.orElseThrow` *throws* a `StudentNotFoundException` with a descriptive message.

8. **`addStudent` Method:** (implemente logging)
    - The method should already exist, annotated with `@POST` and `@Consumes(MediaType.APPLICATION_JSON)`.
    - It should set the ID of the new student and add it to the `students` list.
    - Log an information message: "Added new student with ID: {}" by student ID.
9. **`updateStudent` Method:** (Modify to throw exception)
    - The method should exist with annotations: `@PUT`, `@Path("/{studentId}")`, `@Consumes(MediaType.APPLICATION_JSON)`, and `@PathParam("studentId")`.
    - **Change the Implementation:**
        - Log an information message "PUT request to update student with ID: {}" using `studentId`.
        - Use a `for` loop (with an index `i`) to iterate through the `students` list.
        - Inside the loop, check if the current student's ID matches `studentId`.
            - If they match:
                - Set the `id` of the `updatedStudent` to `studentId`.
                - Replace the existing student at index `i` with `updatedStudent` using `students.set(i, updatedStudent);`.
                - Log an information message: "Updated student with ID: {}" using `studentId`
                - `return;` from the method.
            - If the loop completes *without* finding a match:
                - `throw new StudentNotFoundException("Student with ID " + studentId + " not found for update");`
10. **`deleteStudent` Method:** (Modify to throw exception)
    - The method should exist with annotations: `@DELETE`, `@Path("/{studentId}")`, and `@PathParam("studentId")`.
    - **Change the Implementation:**
        - Log an information message: "DELETE request for student with ID: {}" using `studentId`.

    - Use `students.removeIf(student -> student.getId() == studentId);`. Store the *result* of this operation in a `boolean` variable (e.g., `boolean removed`). `removeIf` returns `true` if an element was removed, and `false` otherwise.
    - Add an `if` statement: `if (!removed) { ... }`
        - Inside the `if` block, `throw new StudentNotFoundException("Student with ID " + studentId + " not found for deletion");`
        - Log information message "Deleted student with ID: {}" using studentId.

11. **`getModulesForStudent` Method:** (Modify to use `orElseThrow` and handle potential `ModuleNotFoundException`)
    - This method already exists, and returns a String. Keep the existing annotations: `@GET`, `@Path("/{studentId}/modules")`, `@Produces(MediaType.APPLICATION_JSON)`, and `@PathParam("studentId")`.
    - **Change the Implementation:**
        - Log an informational message "GET request for modules for student with ID: {}" using studentId.
        - Call `getStudentById(studentId)` to retrieve the student. Because getStudentById now throws an exception, we don't need the null check in the original code.
    - `int moduleId = 1; // Assuming the student selected module with ID 1`
    -
    - Retrieve the module using stream. Call `ModuleResource.getAllModulesStatic()` use `.stream()` to get the stream.
    - Call `filter` with the condition to filter modules with moduleId.
    - Call `findFirst()` and `orElseThrow()` and throw `ModuleNotFoundException` passing "Module with ID " + moduleId + " not found." as the parameter
    - Check if the `selectedModule` is not null
        - Log an informational message ""Retrieved module '{}' for student ID: {}" with the selected module's name and the studentId.
        - Create JSON formatted string and return.
    - Otherwise throw ModuleNotFoundException with message "Module with ID " + moduleId + " not found."

NOTES:

1. **`students.stream():`**
   - This line assumes `students` is a collection (e.g., a `List<Student>`) containing `Student` objects.
   - The `.stream()` method creates a stream from this collection, allowing us to perform functional-style operations on the students.
2. **`.filter(student -> student.getId() == studentId):`**
   - This is a filtering operation.
   - It takes a lambda expression `student -> student.getId() == studentId` as a predicate.
   - For each `student` in the stream, it checks if `student.getId()` is equal to the provided `studentId`.
   - Only students whose IDs match `studentId` will pass through this filter and remain in the stream.
3. **`.findFirst():`**
   - This is a terminal operation that attempts to find the first element in the filtered stream.
   - If a student with the matching ID is found, `findFirst()` will return an `Optional<Student>` containing that student.
   - If no student with the matching ID is found, `findFirst()` will return an empty `Optional<Student>`.
4. **`.orElseThrow(() -> new StudentNotFoundException("Student with ID " + studentId + " not found")):`**
   - This is another terminal operation that handles the `Optional<Student>` returned by `findFirst()`.
   - `.orElseThrow()` takes a supplier (a function that provides a value) as an argument.
   - If the `Optional<Student>` is present (meaning a student was found), `.orElseThrow()` will return the student object.
   - If the `Optional<Student>` is empty (meaning no student was found), `.orElseThrow()` will execute the supplier, which in this case creates and throws a `StudentNotFoundException`.
   - The `StudentNotFoundException` is a custom exception that provides a descriptive error message indicating that the student with the specified ID was not found.

In essence, this code does the following:

- It searches for a student with a specific `studentId` within a collection of `students`.
- If a student with the matching ID is found, it returns that student.
- If no student with the matching ID is found, it throws a `StudentNotFoundException` with an appropriate error message.

Benefits of this approach:

- **Readability:** The code is concise and easy to understand, thanks to the use of streams and lambda expressions.
- **Safety:** The `orElseThrow()` method ensures that the code handles the case where the student is not found, preventing potential `NullPointerException` errors.
- **Efficiency:** Streams can be optimized by the Java runtime, potentially leading to improved performance, especially with large collections.
- **Functional style:** promotes cleaner and more maintainable code.

TEST THE API

- Please test the API using ReqBin or POSTMAN and share your results with your instructor.

TEACHER AND MODULE RESOURCE CLASS (INDEPENDENT EXERCISE)

- Please apply the same changes that we did in the student class to complete teacher and resource classes.