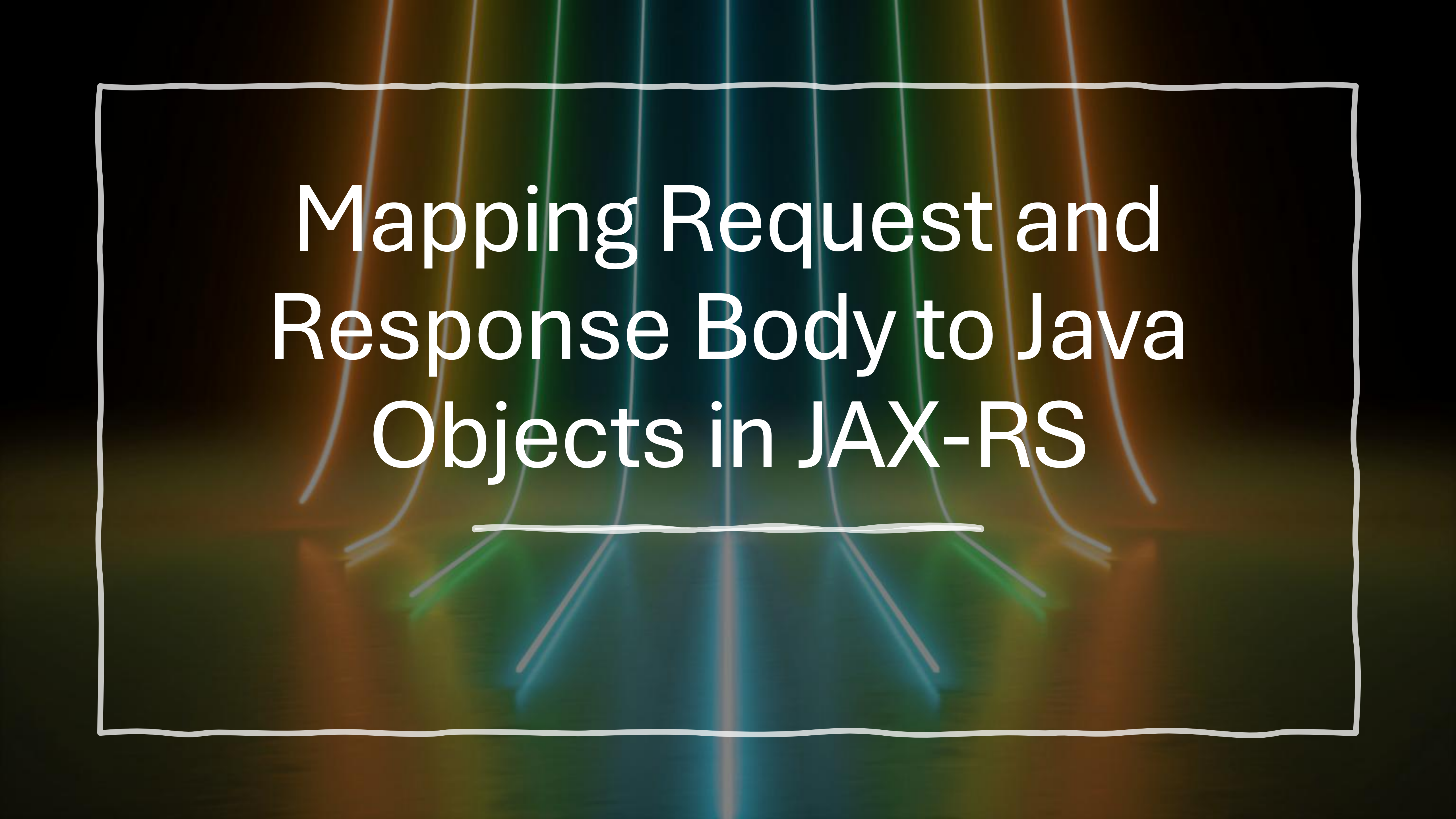
Lecture Week 07
Advanced JAX-RS

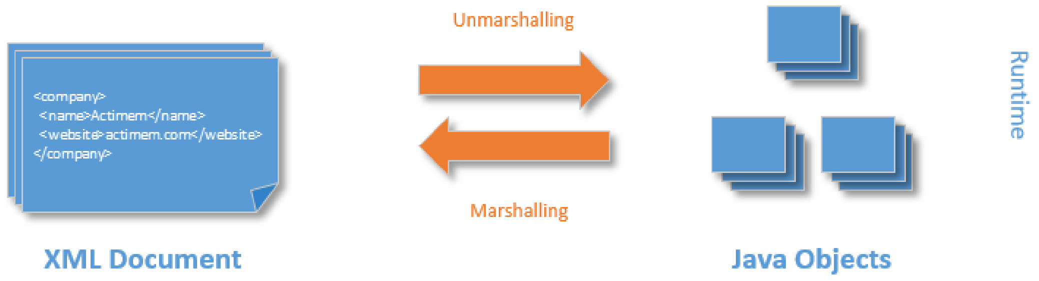
Dr. Hamed Hamzeh



Mapping Request and Response Body to Java Objects in JAX-RS

marshalling & unmarshalling

- In **JAX-RS**, **marshalling** and **unmarshalling** refer to the processes of converting Java objects to JSON or XML format (serialization) and vice versa (deserialization).



```
<company>
 <name>Actimem</name>
 <website>actimem.com</website>
</company>
```

Jackson

- Jackson provides a set of data-binding annotations and a powerful ObjectMapper class to convert JSON data to Java objects (serialization) and vice versa (deserialization).

```
import com.fasterxml.jackson.databind.ObjectMapper;
```

Jackson dependencies

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.12.5</version> <!-- Adjust version as needed -->
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.12.5</version> <!-- Adjust version as needed -->
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>2.12.5</version> <!-- Adjust version as needed -->
</dependency>
```

```java
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonExample {
    public static void main(String[] args) throws Exception {
        // Creating an instance of the ObjectMapper
        ObjectMapper objectMapper = new ObjectMapper();

        // Creating a sample Java object
        Person person = new Person("John Doe", 25);

        // Serialization: Converting Java object to JSON
        String json = objectMapper.writeValueAsString(person);
        System.out.println("Serialized JSON: " + json);

        // Deserialization: Converting JSON to Java object
        Person deserializedPerson = objectMapper.readValue(json, Person.class);
        System.out.println("Deserialized Person: " + deserializedPerson);
    }
}

// Sample Java class for demonstration
class Person {
    private String name;
    private int age;

    // Constructors, getters, and setters (not shown for brevity)

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

Exception Handling in JAX-RS

The Problem: Unhandled Exceptions

Without specific handling, exceptions in your JAX-RS resources can lead to:

- Generic, unhelpful 500 Internal Server Error responses.Leaking internal implementation details (stack traces, etc. - a security risk!).
- Poor user experience for clients of your API.
- Difficult debugging.

Clients need meaningful, consistent error responses.

The Solution: ExceptionMapper

ExceptionMapper is a JAX-RS interface.

It allows you to map specific exceptions to custom Response objects.

Provides centralized, consistent exception handling logic.

Gives you fine-grained control over HTTP status codes, error messages, and response bodies.

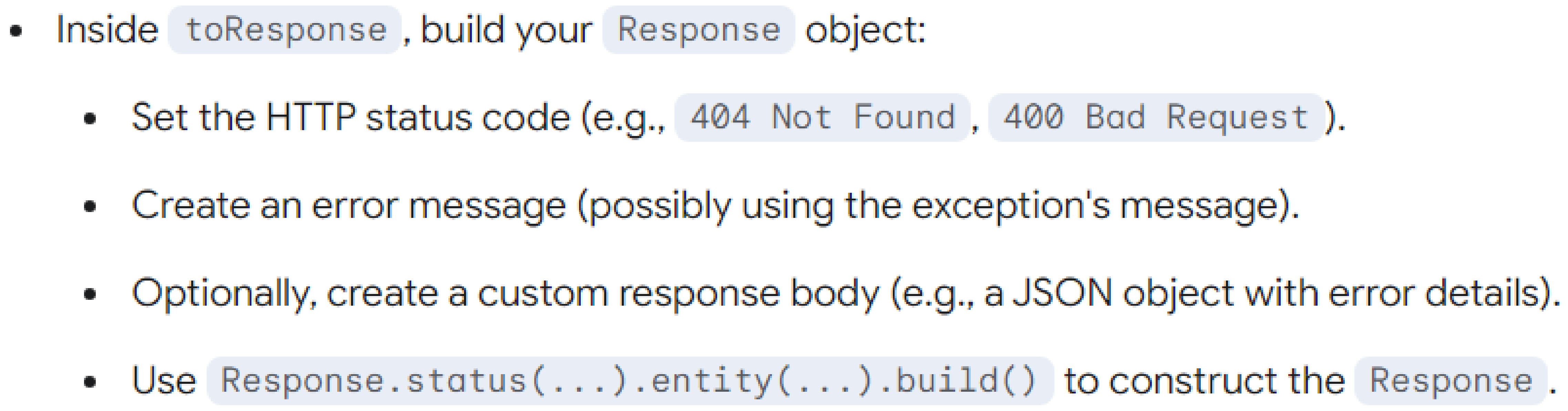
Define Your Exception

- Often, you'll create custom exception classes to represent specific error conditions in your application.
- This makes your code more readable and maintainable.
- Example: `ProductNotFoundException`, `InvalidInputException`, `OrderProcessingException`
- *Not strictly required* – you can map built-in Java exceptions too.

Implement the ExceptionMapper

- Create a class that implements `javax.ws.rs.ext.ExceptionMapper<YourException>`.
  - Replace `YourException` with the exception you want to handle (e.g., `ProductNotFoundException`).
- Annotate the class with `@Provider`. This makes JAX-RS discover it.
- Implement the `toResponse(YourException exception)` method.
  - This method takes the exception as input.
  - It *returns* a `javax.ws.rs.core.Response` object.

Crafting the Response (Inside toResponse)



- Inside `toResponse`, build your `Response` object:
  - Set the HTTP status code (e.g., `404 Not Found`, `400 Bad Request`).
  - Create an error message (possibly using the exception's message).
  - Optionally, create a custom response body (e.g., a JSON object with error details).
  - Use `Response.status(...).entity(...).build()` to construct the `Response`.

Register the Provider (If Necessary)

- `@Provider` annotation usually enough for auto-discovery in many JAX-RS implementations (like Jersey, RESTEasy).
- In some environments (e.g., older Servlet containers without scanning), you may need to explicitly register your `ExceptionMapper` in your application's configuration (e.g., `web.xml` or a JAX-RS `Application` subclass).
- Check your specific JAX-RS implementation's documentation for details.

Code Example: ProductNotFoundException

```
package com.example.exceptions;


// Step 1: Custom Exception (Optional, but good practice)
public class ProductNotFoundException extends RuntimeException {
    public ProductNotFoundException(String message) {
        super(message);
    }
}
```

Code Example: ProductNotFoundException

```
import com.example.exceptions.ProductNotFoundException;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;
import javax.ws.rs.core.MediaType;

// Step 2: Implement ExceptionMapper
@Provider
public class ProductNotFoundExceptionMapper implements
ExceptionMapper<ProductNotFoundException> {

    // Step 3: Craft the Response
    @Override
    public Response toResponse(ProductNotFoundException exception) {
        return Response.status(Response.Status.NOT_FOUND) // 404
                .entity("{\"error\": \"Product not found\", \"message\": \"" +
exception.getMessage() + "\"}") // Custom JSON body
                .type(MediaType.APPLICATION_JSON)
                .build();
    }
}
```

Code Example: ProductNotFoundException

```
import com.example.exceptions.ProductNotFoundException;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/products")
@Produces(MediaType.APPLICATION_JSON)
public class ProductResource {

    @GET
    @Path("/{id}")
    public String getProduct(@PathParam("id") int id) {
        // Simulate fetching a product; throw exception if not found
        if (id != 123) {
            throw new ProductNotFoundException("Product with ID " + id + " not found.");
        }
        return "{\"id\": 123, \"name\": \"Example Product\"}";
    }
}
```

Beyond the Basics

- Mapping Multiple Exceptions: You can create multiple ExceptionMapper classes.
- Generic ExceptionMapper: Create a mapper for Throwable to catch all unhandled exceptions (as a last resort). Useful for logging unexpected errors.
- Exception Hierarchy: JAX-RS will use the most specific ExceptionMapper that matches the thrown exception.
- Context Injection: You can inject @Context objects (like HttpServletRequest) into your ExceptionMapper if you need request-specific information.

Asynchronous RESTful Web Services

Asynchronous RESTful web services use asynchronous communication patterns within the REST architecture.

Traditional REST relies on synchronous communication where the client waits for a response.

In the Asynchronous Model the client doesn't wait for an immediate response, improving efficiency.

Use Asynchronous Processing in Resource Methods

We can define a JAX-RS resource class with a method that supports asynchronous processing.

We use the **@Suspend** annotation to indicate that the method supports asynchronous operations.

Additionally, the method includes a parameter of type **AsyncResponse**, which allows us to control the asynchronous response.

Example

```
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/asyncResource")
public class AsyncResource {

    @GET
    @Path("/example")
    public void asyncMethod(@Suspended AsyncResponse asyncResponse) {
        // Your asynchronous processing logic here
        // Use asyncResponse to control the response
    }
}
```

Processing Asynchronous Response

```java
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/asyncResource")
public class AsyncResource {

    @GET
    @Path("/example")
    public void asyncMethod(@Suspended AsyncResponse asyncResponse) {
        // Simulate asynchronous processing
        new Thread(() -> {
            try {
                Thread.sleep(5000); // Simulating a time-consuming operation
                asyncResponse.resume("Asynchronous response data");
            } catch (InterruptedException e) {
                asyncResponse.resume(e);
            }
        }).start();
    }
}
```

Exception Handling

```
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/asyncResource")
public class AsyncResource {

  @GET
  @Path("/example")
  public void asyncMethod(@Suspended AsyncResponse asyncResponse) {
    // Simulate asynchronous processing
    new Thread(() -> {
      try {
        Thread.sleep(5000); // Simulating a time-consuming operation
        asyncResponse.resume("Asynchronous response data");
      } catch (InterruptedException e) {
        asyncResponse.resume(e); // Resume with exception for error handling
      }
    }).start();
  }
}
```

Logging

Log4j

- Log4j is a reliable, flexible, and fast logging framework written in Java by Apache Software Foundation.
- It is used for outputting log statements from applications to various output targets.

```java
import org.apache.log4j.Logger;

public class Log4jExample {
    private static Logger log =
Logger.getLogger(Log4jExample.class);

    public static void main(String[] args) {
        log.info("This is an info message");
        log.error("This is an error message");
    }
}
```

SLF4J

- The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks.
- It allows the end-user to plug in the desired logging framework at deployment time.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Slf4jExample {
    private static Logger log =
LoggerFactory.getLogger(Slf4jExample.class);

    public static void main(String[] args) {
        log.info("This is an info message");
        log.error("This is an error message");
    }
}
```

Table

Criteria	Log4j	SLF4J
Performance	Known for its speed and is faster than java.util.logging.	Just a facade and its performance depends on the underlying logging framework used.
Flexibility	Offers high flexibility in formatting logs and provides a slightly more sophisticated configuration setup.	Offers high flexibility in formatting logs.
Compatibility	Can be used directly in your application.	Provides an abstraction layer over other logging frameworks, making it a better choice if you want to switch between different logging frameworks.
Community Support	Has strong community support and is widely used in the industry.	Has strong community support and is widely used in the industry.

Using Loggers in JAX-RS

- First, you need to add the necessary dependencies to your project.
- If you're using Maven, add the following to your pom.xml:

```xml
<dependencies>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>2.x.x</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.x.x</version>
    </dependency>
</dependencies>
```

Filters in JAX-RS

Why do we need filters?

- **Modifying Request or Response Parameters**: Filters can be used to modify request or response parameters like headers
- **Authentication**: Filters can be used for things like authentication. For example, you can use a filter to check if a request has a valid authentication token.
- **Logging and Auditing**: Filters can be used for logging and auditing. For example, you can use a filter to log all incoming requests.
- **Data Conversion/Transformation/Compression**: Filters can be used for data conversion, transformation, and compression. For example, you can use a filter to compress the response data before sending it to the client.
- **Localization Targeting**: Filters can be used for localization targeting. For example, you can use a filter to determine the user's locale and customize the response accordingly.

First Step in Creating Filters

First, you need to create a class that implements either:

ContainerRequestFilter for modifying request parameters or

ContainerResponseFilter for modifying response parameters.

In this class, you override the **filter** method where you can access and modify the headers.

Example

```
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.ext.Provider;

@Provider
public class CustomHeaderFilter implements ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext, ContainerResponseContext
responseContext) {
        responseContext.getHeaders().add("X-Custom-Header", "This is a custom header");
    }
}
```

Second Step in Creating Filters

- The next step is to register the filter.
- If you’re using a **@Provider** annotation (as in the previous example), the JAX-RS runtime will automatically discover and register the filter.

```
public class MyApplication extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<>();
        classes.add(MyResource.class);
        classes.add(CustomHeaderFilter.class);
        return classes;
    }
}
```

Exception Filters

- **ExceptionMapper**:
  - Handles exceptions thrown during the processing of a request or response.
  - It maps exceptions to appropriate HTTP responses, providing a way to customize error handling.

```
@Provider
public class CustomExceptionMapper implements ExceptionMapper<CustomException> {

    @Override
    public Response toResponse(CustomException exception) {
        return Response.status(Response.Status.INTERNAL_SERVER_ERROR)
                       .entity("Error: " + exception.getMessage())
                       .build();
    }
}
```

JPA

- JPA stands for Java Persistence API. It is a specification for object-relational mapping (ORM) in Java.
- JPA provides a set of interfaces and annotations that define how Java objects are mapped to relational database tables and how they are persisted, retrieved, and managed.
- JPA allows developers to write Java code that interacts with databases without having to write SQL queries manually.
- Instead, JPA uses an ORM framework to translate Java objects into SQL queries, and vice versa.
- JPA is part of the Java EE (Enterprise Edition) platform, but it can also be used in Java SE (Standard Edition) applications.

JPA

javax.persistence.Entity	is used to mark a Java class as a persistent entity. In other words, it indicates that an instance of this class represents a persistent entity in a database.
javax.persistence.Id	This annotation is used to indicate that the field or property is to be used as the unique identifier for the entity, and is typically used in conjunction with other annotations such as @Entity, @Table, and @GeneratedValue.
javax.persistence.GenerationType	is an enum in the Java Persistence API (JPA) that specifies how the primary key value for an entity is generated.
javax.persistence.OneToMany	is a relationship mapping between two database tables/entities in Java Persistence API (JPA).
javax.persistence.CascadeType	In JPA, when you have a relationship between two entities, changes to one entity can affect the other entity. The cascade type defines the behavior of how these changes should propagate from one entity to the other.

DAO in Java

- DAO (Data Access Object) is a design pattern that provides an abstraction layer between the **business logic** and the **data persistence layer** (usually a database).
- The DAO pattern allows developers to decouple the code responsible for interacting with the database from the rest of the application.
- This separation of concerns enables developers to modify the database layer independently of the application logic, without affecting the rest of the code.
- A DAO interface typically defines a set of methods for performing CRUD (Create, Read, Update, Delete) operations on a particular entity or group of entities.
- The concrete implementation of the DAO interface interacts with the database to perform these operations.

Example

```java
public class Book {
    private Long id;
    private String title;
    private String author;

    // getters and setters
}
```

```java
import java.util.List;

public interface BookDAO {
    List<Book> getAllBooks();
    Book getBookById(Long id);
    void addBook(Book book);
    void updateBook(Book book);
    void deleteBook(Long id);
}
```

```java
import java.util.ArrayList;
import java.util.List;

public class InMemoryBookDAO implements BookDAO {
    private List<Book> books = new ArrayList<>();
    private Long nextId = 1L;

    @Override
    public List<Book> getAllBooks() {
        return new ArrayList<>(books);
    }

    @Override
    public Book getBookById(Long id) {
        for (Book book : books) {
            if (book.getId().equals(id)) {
                return book;
            }
        }
        return null;
    }

    @Override
    public void addBook(Book book) {
        book.setId(nextId);
        nextId++;
        books.add(book);
    }

    @Override
    public void updateBook(Book updatedBook) {
        for (int i = 0; i < books.size(); i++) {
            Book book = books.get(i);
            if (book.getId().equals(updatedBook.getId())) {
                books.set(i, updatedBook);
                return;
            }
        }
    }

    @Override
    public void deleteBook(Long id) {
        books.removeIf(book -> book.getId().equals(id));
    }
}
```

```java
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import java.util.List;

@Path("/books")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class BookResource {
    private BookDAO bookDAO = new InMemoryBookDAO();

    @GET
    public List<Book> getAllBooks() {
        return bookDAO.getAllBooks();
    }

    @GET
    @Path("/{id}")
    public Book getBookById(@PathParam("id") Long id) {
        return bookDAO.getBookById(id);
    }

    @POST
    public void addBook(Book book) {
        bookDAO.addBook(book);
    }

    @PUT
    public void updateBook(Book book) {
        bookDAO.updateBook(book);
    }

    @DELETE
    @Path("/{id}")
    public void deleteBook(@PathParam("id") Long id) {
        bookDAO.deleteBook(id);
    }
}
```

Best Practices for RESTful service design

Match HTTP methods to CRUD operations

GET: Retrieve resource representation or a collection of resources.	Example: GET /users retrieves a list of users.
POST: Create a new resource or perform a non-idempotent operation.	Example: POST /users creates a new user.
PUT: Update an existing resource or create a new resource if it doesn't exist (idempotent).	Example: PUT /users/123 updates user with ID 123.
DELETE: Remove a resource.	Example: DELETE /users/123 deletes user with ID 123.

Proper use of HTTP status codes

200 OK:

- Successful GET request.

201 Created:

- Successful POST request resulting in the creation of a new resource. Include Location header with the URI of the newly created resource.

204 No Content:

- Successful request, but there is no representation to return (common for DELETE operations).

404 Not Found:

- Resource not found. Include details in the response body.

400 Bad Request:

- Malformed request. Include details in the response body for better understanding of the issue.

URI Design

Choose meaningful and intuitive URIs:

- Select URIs that are easy to understand and reflect the nature of the resource.
- Example: /users for a collection of users, /users/{id} for a specific user.

Follow the hierarchy for resource representation:

- Organize URIs hierarchically to represent the relationship between resources.
- Example: /departments/123/employees to represent employees in department 123.

URI design

Use nouns for resources, and avoid verbs in URIs:

- Use nouns to represent resources, making URIs more expressive.
- Example: /orders instead of /getOrders, /users instead of /retrieveUsers.

Consider versioning in URIs for future changes:

- Plan for API versioning to accommodate future changes without breaking existing clients.
- Example: /v1/users or /v2/users to denote different versions of the user resource.