Web Applications Developments

ITWT413

LECTURE 6: JAKARTA EE CORE PROFILE

DR. HALA SHAARI

Course Structure

- Lecture (Monday& Tuesday)
- Time: 12:30-14:00
- Microsoft Teams Code (goi2d67)

(Lectures, labs, announcements, References):

- Grading :
  - 25% Midterm exam.
  - 25% Assignments
  - 25% Group Project (Groups of two).
  - 25% Final Exam.

Course References

Our Course Main References:

- Jakarta EE Tutorial
- Java EE to Jakarta EE 10 Recipes: A Problem-Solution Approach for Enterprise Java (2022)

key course objectives

1) Master Core Jakarta EE Concepts: Understand the fundamental architecture and components of Jakarta EE, including Servlets, JSP, JSF, and Enterprise JavaBeans (EJB).
2) Develop RESTful Web Services: Build and deploy scalable RESTful APIs using Jakarta RESTful Web Services (JAX-RS), with advanced features like exception handling, filters, and security.
3) Implement Dependency Injection and Persistence: Leverage Contexts and Dependency Injection (CDI) and Jakarta Persistence (JPA) to manage beans and database interactions in enterprise applications.
4) Ensure Application Security and Transaction Management: Apply Jakarta EE's security framework for authentication, authorization, and manage transactions using declarative and programmatic approaches.
5) Deploy Cloud-Native Applications: Utilize Jakarta EE to build, containerize, and deploy microservice-based applications in cloud environments, incorporating tools like Docker and Kubernetes.

Lecture Agenda

- Building RESTful Web Services with Jakarta REST
  - What Are RESTful Web Services?
  - Creating a RESTful Root Resource Class
  - Developing RESTful Web Services with Jakarta REST
  - Overview of a Jakarta REST Application
  - The @Path Annotation and URI Path Templates
  - Responding to HTTP Methods and Requests
  - Using @Consumes and @Produces to Customize Requests and Responses
  - Extracting Request Parameters
  - Configuring Jakarta REST Applications
  - Example Applications for Jakarta REST

Required Software

The following software is required to run the examples

- Java Platform, Standard Edition
- Eclipse Glassfish Server
- Jakarta EE Tutorial Examples
- Apache NetBeans IDE
- Apache Maven
- Instructions from page 11 to page17 from Jakarta EE Tutorial tells you everything you need to know to install, build, and run the tutorial examples

Jakarta EE Tutorial- Structure

Basic Platform	Jakarta EE Core Profile	Jakarta EE Web Profile	Jakarta EE Platform
Resource Creation	Jakarta CDI Lite	Jakarta CDI Full	Jakarta Mail
Injection	Jakarta REST	Jakarta Validation	Jakarta Messaging
Packaging	Jakarta JSON	Jakarta Security	Jakarta Batch
		Jakarta Servlets	
		Jakarta Faces	
		Jakarta WebSocket	
		Jakarta Persistence	
		Jakarta Enterprise Beans Lite	

What Are RESTful Web Services?

- RESTful web services are loosely coupled, lightweight web services that are particularly well suited for creating APIs for clients spread out across the internet.
- Representational State Transfer (REST) is an architectural style of client-server application centered around the transfer of representations of resources through requests and responses. In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links on the Web. The resources are represented by documents and are acted upon by using a set of simple, well-defined operations.
- For example, a REST resource might be the current weather conditions for a city. The representation of that resource might be an XML document, an image file, or an HTML page. A client might retrieve a particular representation, modify the resource by updating its data, or delete the resource entirely.
- The REST architectural style is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

Principles for to be simple, lightweight, and fast RESTful applications (1)

- Resource identification through URI: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery.
- Uniform interface: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource.

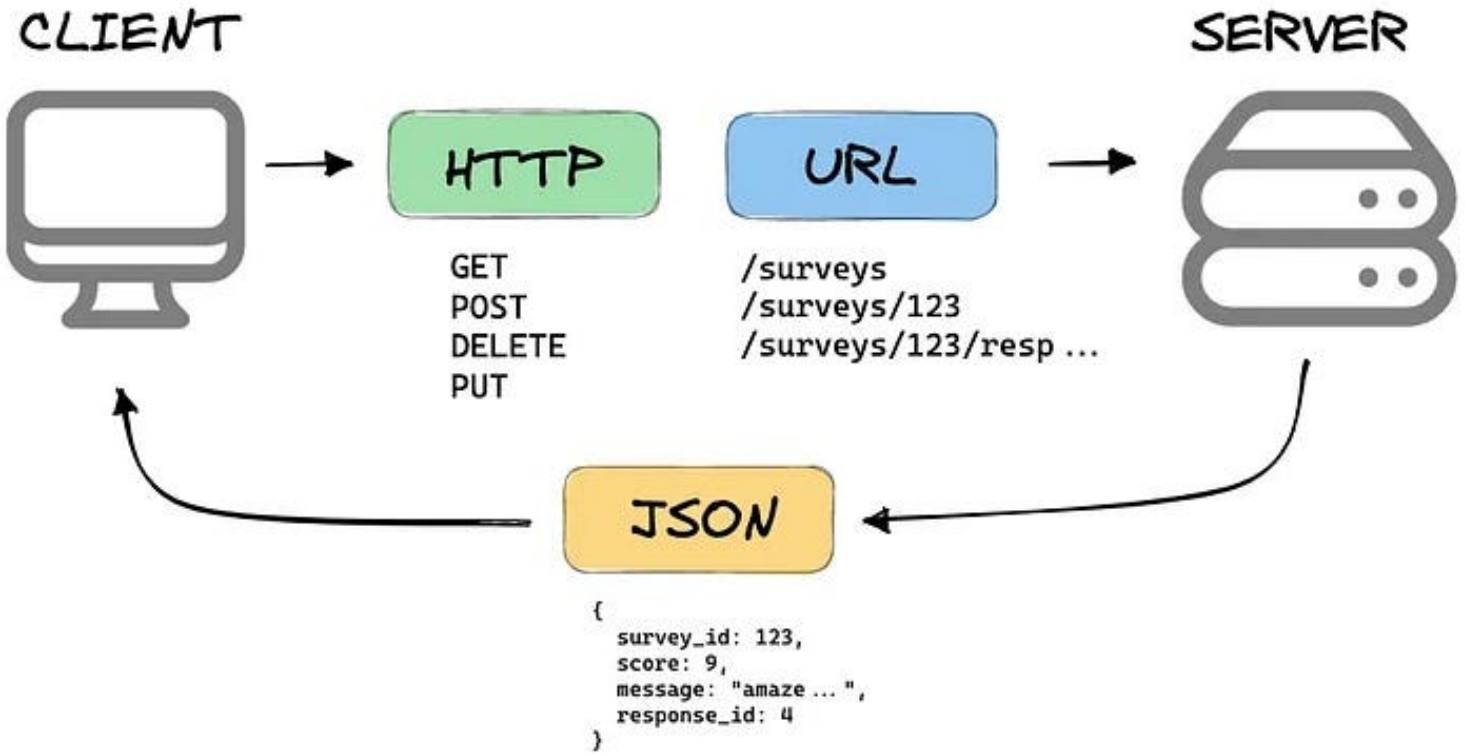
Principles for to be simple, lightweight, and fast RESTful applications (2)

- Self-descriptive messages: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and other document formats. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.
- Stateful interactions through links: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction.

Creating a RESTful Root Resource Class

- Root resource classes are "plain old Java objects" (POJOs) that are either annotated with @Path or have at least one method annotated with @Path or a request method designator, such as @GET, @PUT, @POST, or @DELETE.
- Resource methods are methods of a resource class annotated with a request method designator.

WHAT IS A REST API?



Developing RESTful Web Services with Jakarta REST

This section explains how to use Jakarta REST to annotate Java classes to create RESTful web services.

- The Jakarta REST API uses Java programming language annotations to simplify the development of RESTful web services.
- Developers decorate Java programming language class files with Jakarta REST annotations to define resources and the actions that can be performed on those resources.
- Jakarta REST annotations are runtime annotations; therefore, runtime reflection will generate the helper classes and artifacts for the resource.
- A Jakarta EE application archive containing Jakarta REST resource classes will have the resources configured, the helper classes and artifacts generated, and the resource exposed to clients by deploying the archive to a Jakarta EE server.

Summary of Jakarta REST Annotations lists (1)

Annotation	Description
`@Path`	The `@Path` annotation's value is a relative URI path indicating where the Java class will be hosted: for example, `/helloworld`. You can also embed variables in the URIs to make a URI path template. For example, you could ask for the name of a user and pass it to the application as a variable in the URI: `/helloworld/{username}`.
`@GET`	The `@GET` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@POST`	The `@POST` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP POST requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@PUT`	The `@PUT` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PUT requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@DELETE`	The `@DELETE` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP DELETE requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.

Summary of Jakarta REST Annotations lists (2)

`@OPTIONS`	The `@OPTIONS` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP OPTIONS requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@PATCH`	The `@PATCH` annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PATCH requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
`@PathParam`	The `@PathParam` annotation is a type of parameter that you can extract for use in your resource class. URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the `@Path` class-level annotation.
`@QueryParam`	The `@QueryParam` annotation is a type of parameter that you can extract for use in your resource class. Query parameters are extracted from the request URI query parameters.
`@Consumes`	The `@Consumes` annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.
`@Produces`	The `@Produces` annotation is used to specify the MIME media types of representations a resource can produce and send back to the client: for example, `"text/plain"`.

Summary of Jakarta REST Annotations lists (3)

`@Provider`	The `@Provider` annotation is used for anything that is of interest to the Jakarta REST runtime, such as `MessageBodyReader` and `MessageBodyWriter`. For HTTP requests, the `MessageBodyReader` is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using a `MessageBodyWriter`. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a `Response` that wraps the entity and that can be built using `Response.ResponseBuilder`.
`@ApplicationPath`	The `@ApplicationPath` annotation is used to define the URL mapping for the application. The path specified by `@ApplicationPath` is the base URI for all resource URIs specified by `@Path` annotations in the resource class. You may only apply `@ApplicationPath` to a subclass of `jakarta.ws.rs.core.Application`.

Overview of a Jakarta REST Application

```java
package ee.jakarta.tutorial.hello;

import jakarta.ws.rs.Consumes;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.PUT;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.Context;
import jakarta.ws.rs.core.UriInfo;

/**
 * Root resource (exposed at "helloworld" path)
 */
@Path("helloworld")
public class HelloWorld {
    @Context
    private UriInfo context;

    /** Creates a new instance of HelloWorld */
    public HelloWorld() {
    }

    /**
     * Retrieves representation of an instance of helloWorld.HelloWorld
     * @return an instance of java.lang.String
     */
    @GET
    @Produces("text/html")
    public String getHtml() {
        return "<html lang=\"en\"><body><h1>Hello, World!!</h1></body></html>";
    }
}
```

Jakarta REST Application Explanation

- The @Path annotation’s value is a relative URI path. In the preceding example, the Java class will be hosted at the URI path /helloworld. This is an extremely simple use of the @Path annotation, with a static URI path. Variables can be embedded in the URIs. URI path templates are URIs with variables embedded within the URI syntax.
- The @GET annotation is a request method designator, along with @POST, @PUT, @DELETE, and @HEAD, defined by Jakarta REST and corresponding to the similarly named HTTP methods. In the example, the annotated Java method will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
- The @Produces annotation is used to specify the MIME media types a resource can produce and send back to the client. In this example, the Java method will produce representations identified by the MIME media type "text/html".
- The @Consumes annotation is used to specify the MIME media types a resource can consume that were sent by the client.

The @Path Annotation and URI Path Templates

- The @Path annotation identifies the URI path template to which the resource responds and is specified at the class or method level of a resource.
- The @Path annotation's value is a partial URI path template relative to the base URI of the server on which the resource is deployed, the context root of the application, and the URL pattern to which the Jakarta REST runtime responds.
- URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by braces ({ and }).
- By default, the URI variable must match the regular expression "[^/]+?". This variable may be customized by specifying a different regular expression after the variable name

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")
```

Responding to HTTP Methods and Requests

- The behavior of a resource is determined by the HTTP methods (typically, GET, POST, PUT, or DELETE) to which the resource is responding.

1) The Request Method Designator Annotations.

   Request method designator annotations are runtime annotations, defined by Jakarta REST, that correspond to the similarly named HTTP methods. Within a resource class file, HTTP methods are mapped to Java programming language methods by using the request method designator annotations. The behavior of a resource is determined by which HTTP method the resource is responding to. Jakarta REST defines a set of request method designators for the common HTTP methods GET, POST, PUT, DELETE, and HEAD; you can also create your own custom request method designators.

2) Using Entity Providers to Map HTTP Response and Request Entity Bodies.

   Entity providers supply mapping services between representations and their associated Java types. The two types of entity providers are MessageBodyReader and MessageBodyWriter. For HTTP requests, the MessageBodyReader is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using a MessageBodyWriter. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a Response that wraps the entity and that can be built by using Response.ResponseBuilder.

Using @Consumes and @Produces to Customize Requests and Responses

- The information sent to a resource and then passed back to the client is specified as a MIME media type in the headers of an HTTP request or response. You can specify which MIME media types of representations a resource can respond to or produce by using the following annotations:
  - jakarta.ws.rs.Consumes
  - jakarta.ws.rs.Produces
- By default, a resource class can respond to and produce all MIME media types of representations specified in the HTTP request and response headers.

Extracting Request Parameters

- Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. A previous example presented the use of the @PathParam parameter to extract a path parameter from the path component of the request URL that matched the path declared in @Path.
- You can extract the following types of parameters for use in your resource class:
  - Query
  - URI path
  - Form
  - Cookie
  - Header
  - Matrix

Configuring Jakarta REST Applications

- A Jakarta REST application consists of at least one resource class packaged within a WAR file. The base URI from which an application's resources respond to requests can be set one of two ways:
  - Using the @ApplicationPath annotation in a subclass of jakarta.ws.rs.core.Application packaged within the WAR
  - Using the servlet-mapping tag within the WAR's web.xml deployment descriptor

Questions?