Web Applications Developments

ITWT413

LECTURE 4: JAKARTA EE CORE PROFILE

DR. HALA SHAARI

Course Structure

- Lecture (Sunday& 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

- Required Software
- Jakarta EE Core Profile
  - Jakarta CDI Lite
  - Jakarta REST
  - Jakarta JSON

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	

Lecture 3 Recap

- A resource is a program object that provides connections to such systems as database servers and messaging systems.
  - Java Naming and Directory Interface (JNDI) naming service enables components to locate other components and resources
  - You inject resources by using the @Resource annotation in an application.
  - You can use a deployment descriptor to override the resource mapping that you specify in an annotation.
- To store, organize, and retrieve data, most applications use a relational database. Jakarta EE components may access relational databases through the JDBC API.
  - DataSource objects that implement connection pooling also produce a connection to the particular data source that the DataSource class represents.
  - A JDBC connection pool is a group of reusable connections for a particular database.
- An application can include a glassfish-resources.xml file that can be used to define resources for that application and others.
- Resource injection enables you to inject any resource available in the JNDI namespace into any container-managed object, such as a servlet, an enterprise bean, or a managed bean.
- Dependency injection enables you to turn regular Java classes into managed objects and to inject them into any other managed object.
- The Main Differences between Resource Injection and Dependency Injection.
- A Jakarta EE application is delivered in a Java Archive (JAR) file, a Web Archive (WAR) file, or an Enterprise Archive (EAR) file.

Jakarta CDI Lite

- Contexts and Dependency Injection (CDI) enables your objects to have their dependencies provided to them automatically, instead of creating them or receiving them as parameters. CDI also manages the lifecycle of those dependencies for you.
- CDI is a set of services that, used together, make it easy for developers to use enterprise beans along with Jakarta Faces technology in web applications.
- Designed for use with stateful objects, CDI also has many broader uses, allowing developers a great deal of flexibility to integrate various kinds of components in a loosely coupled but typesafe way.
- For related specifications see the Jakarta EE Tutorial

CDI Lite Simple Example(1)

JAVA |

```java
@WebServlet("/cdiservlet")
public class NewServlet extends HttpServlet {
    private Message message;

    @Override
    public void init() {
        message = new MessageB();
    }

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                throws IOException {
        response.getWriter().write(message.get());
    }
}
```

```java
public interface Message {
    public String get();
}
```

The servlet creates itself an instance of the following object:

```java
public class MessageB implements Message {
    public MessageB() { }

    @Override
    public String get() {
        return "message B";
    }
}
```

CDI Lite Simple Example(2)

```
@WebServlet("/cdiservlet")
public class NewServlet extends HttpServlet {
    @Inject private Message message;

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
                throws IOException {
        response.getWriter().write(message.get());
    }
}
```

```
@RequestScoped
public class MessageB implements Message { ... }
```

CDI Lite vs CDI Full

- CDI Lite provides a subset of the CDI Full functionality with an emphasis on build-time implementations. By leaving out the additional components from CDI Full such as those dealing with runtime reflection, CDI Lite is able to execute in lighter and more restricted environments.

- Jakarta EE-compliant application servers will still implement the CDI Full functionality so this change will benefit those developers working in alternate (e.g. cloud-based) environments without affecting those working in a standard Jakarta EE environment.

- For more details about functionality available only in CDI Full, see Jakarta EE Tutorial.

Beans & CDI Lite

- CDI redefines the concept of a bean beyond its use in other Java technologies, such as the JavaBeans and Jakarta Enterprise Beans technologies.
- In CDI, a bean is a source of contextual objects that define application state or logic.
- A Jakarta EE component is a bean if the lifecycle of its instances may be managed by the container according to the lifecycle context model defined in the CDI specification.
- The concept of injection has been part of Java technology for some time.
- Since the Java EE 5 platform was introduced, annotations have made it possible to inject resources and some other kinds of objects into container-managed objects.
- CDI makes it possible to inject more kinds of objects and to inject them into objects that are not container-managed.

kinds of objects can be injected

- Almost any Java class
- Session beans
- Jakarta EE resources: data sources, Messaging topics, queues, connection factories, and the like
- Persistence contexts (Jakarta Persistence EntityManager objects)
- Producer fields
- Objects returned by producer methods
- Web service references
- Remote enterprise bean references

Injecting Beans Example(1)

```java
package greetings;

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Target;

import jakarta.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Informal {}
```

```java
package greetings;

@Informal
public class InformalGreeting extends Greeting {
    public String greet(String name) {
        return "Hi, " + name + "!";
    }
}
```

```java
package greetings;

import jakarta.enterprise.inject.Default;

@Default
public class Greeting {
    public String greet(String name) {
        return "Hello, " + name + ".";
    }
}
```

Injecting Beans Example(2)

```
import jakarta.inject.Inject;

public class Printer {

    @Inject Greeting greeting;
    ...
}
```

```
import jakarta.inject.Inject;

public class Printer {

    @Inject @Informal Greeting greeting;
    ...
}
```

Using Scopes

- For a web application to use a bean that injects another bean class, the bean needs to be able to hold state over the duration of the user's interaction with the application.
- The way to define this state is to give the bean a scope.
- You can give an object any of the scopes described in Scopes, depending on how you are using it.
- The first three scopes form the next slide are defined by both Jakarta Context and Dependency Injection and the Jakarta Faces specification.
- The last two are defined by Jakarta Context and Dependency Injection.
- All predefined scopes except @Dependent are contextual scopes.
- CDI places beans of contextual scope in the context whose lifecycle is defined by the Jakarta EE specifications.

Scopes

Scope	Annotation	Duration
Request	`@RequestScoped`	A user's interaction with a web application in a single HTTP request.
Session	`@SessionScoped`	A user's interaction with a web application across multiple HTTP requests.
Application	`@ApplicationScoped`	Shared state across all users' interactions with a web application.
Dependent	`@Dependent`	The default scope if none is specified; it means that an object exists to serve exactly one client (bean) and has the same lifecycle as that client (bean).
Conversation	`@ConversationScoped`	A user's interaction with a servlet, including Jakarta Faces applications. The conversation scope exists within developer-controlled boundaries that extend it across multiple requests for long-running conversations. All long-running conversations are scoped to a particular HTTP servlet session and may not cross session boundaries.

Using Scopes

- You can also define and implement custom scopes, but that is an advanced topic.
- Custom scopes are likely to be used by those who implement and extend the CDI specification.
- A scope gives an object a well-defined lifecycle context.
- A scoped object can be automatically created when it is needed and automatically destroyed when the context in which it was created ends.
- Moreover, its state is automatically shared by any clients that execute in the same context.

More in Scopes

- Jakarta EE components, such as servlets and enterprise beans, and JavaBeans components do not by definition have a well-defined scope. These components are one of the following:
  - Singletons, such as enterprise singleton beans, whose state is shared among all clients
  - Stateless objects, such as servlets and stateless session beans, which do not contain client-visible state
  - Objects that must be explicitly created and destroyed by their client, such as JavaBeans components and stateful session beans, whose state is shared by explicit reference passing between clients.
- However, if you create a Jakarta EE component that is a managed bean, then it becomes a scoped object, which exists in a well-defined lifecycle context.

Giving Beans EL Names

- To make a bean accessible through the EL(unified expression language), use the @Named built-in qualifier.
- The @Named qualifier allows you to access the bean by using the bean name, with the first letter in lowercase. For example, a Facelets page would refer to the bean as printer.
- You can specify an argument to the @Named qualifier to use a nondefault name: @Named("MyPrinter")

```
import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Inject;
import jakarta.inject.Named;

@Named
@RequestScoped
public class Printer {

    @Inject @Informal Greeting greeting;
    ...
}
```

Adding Setter and Getter Methods

- To make the state of the managed bean accessible, add setter and getter methods for that state.
- The createSalutation method calls the bean's greet method, and the getSalutation method retrieves the result.
- Once the setter and getter methods have been added, the bean is complete. The final code looks like this:

JAVA |

```java
package greetings;

import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Inject;
import jakarta.inject.Named;

@Named
@RequestScoped
public class Printer {

    @Inject @Informal Greeting greeting;

    private String name;
    private String salutation;

    public void createSalutation() {
        this.salutation = greeting.greet(name);
    }

    public String getSalutation() {
        return salutation;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Using a Managed Bean in a Facelets Page

- To use the managed bean in a Facelets page, create a form that uses user interface elements to call its methods and to display their results.

```
<h:form id="greetme">
   <p><h:outputLabel value="Enter your name: " for="name"/>
      <h:inputText id="name" value="#{printer.name}"/></p>
   <p><h:commandButton value="Say Hello"
                       action="#{printer.createSalutation}"/></p>
   <p><h:outputText value="#{printer.salutation}"/></p>
</h:form>
```

The simplegreeting CDI Example

- The simplegreeting example illustrates some of the most basic features of CDI:
  - scopes,
  - qualifiers,
  - bean injection,
  - and accessing a managed bean
- in a Jakarta Faces application.
- When you run the example, you click a button that presents either a formal or an informal greeting, depending on how you edited one of the classes.
- The example includes four source files, a Facelets page and template, and configuration files.

The simplegreeting Source Files

- The four source files for the simplegreeting example are:
  - The default Greeting class, shown in Beans as Injectable Objects
  - The @Informal qualifier interface definition and the InformalGreeting class that implements the interface, both shown in Using Qualifiers
  - The Printer managed bean class, which injects one of the two interfaces, shown in full in Adding Setter and Getter Methods
- The source files are located in the jakartaee-examples/tutorial/cdi/simplegreeting/src/main/java/jakarta/tutorial/simplegreeting directory.

Questions?