Contents

Day 1/90

Date : 02/April/2023

Understanding JavaEE - The theory of JavaEE

What is JavaEE?

JavaEE a collection of **abstract** specs, that together form a complete solution, to solve *commonly faced challenges*.

- JavaEE is said to be abstract because; us developers are abstracted away from the implementation, we only code the JavaEE API's that we are given.

  e.g. If we want to pass something to a relational database, we'll just have to call the `entity manager` from the `javax` package. There is no need to worry about what is implementing and what is going on behind the scene.

Some common faced challenges : Persistence, Web services, Transactions, Security, etc ...

-What is an Application Server?

An Implementation of the entire body of JavaEE abstract specifications is called "Application Server".

We are going to use all the implementation from `import javax.*` package (or also the newly `jakrata.*` package).

JavaEE is a set of specifications and standards that provides developers with a set of APIs for building enterprise-level applications. These APIs are *portable*, which means that they can be run on any compatible application server, regardless of the operating system or hardware platform. This ensures that applications built with JavaEE can run on any compatible platform without requiring any changes to the code.

JavaEE is *Portable* ; it means when we develop applications using the standard interfaces, we can then deploy it or we can then run it on any given JavaEE implementation and our application should work.

Examples of JavaEE Application Server

There are a bunch of servers provided to us :

- Payara Server (Glassfish)
- IBM OpenLiberty
- JBOSS Wildfly

all these app servers, implement the abstract specs meaning, when I'm developing, I can use IBM server for testing, and when it was in the stage of publication, I can switch to Glassfish.

If we are saying JavaEE is a collection of abstract specs, than why is there different APIs. For instance why is *Persistence API* different from *Dependency Injection API* ? These are realized with **JSR (Java Specification Request)**.

What is a JSR

JSR stands for "Java Specification Request", which is a formal proposal submitted by members of the Java community to the Java Community Process (JCP) for the development and enhancement of technology and the Java technology platform.

JSR defines a way to introduce new technologies or improvements to existing ones in the Java language, libraries, and frameworks. Each JSR outlines a specific problem that needs to be solved and a proposed solution for that.

> In simple words, these abstract specs are grouped in the form of silos in the form of JSRs.

JSR Examples

JSRs by Platform : All the Java (JavaEE, JavaSE, JavaME) platforms are grouped here.

APIs available on the JavaEE platform : XML Parsing, Enterprise Java Beans (EJB), RESTful Web Service (JAX-RS), etc ...

The app server has already been implemented these specifications. So when we want to create a let's say RESTful GET web service, we can use the annotation or whatever the method is. The app server will run without any errors.

> In conclusion, the JSR specs tell us what we can do with specific API. Like a guide or documentation of the API.

For every JSR, there is a *Reference implementation*.

What is a Reference Implementation?

A complete realization of the abstract JSRs is what is called a Reference Implementation.

For Instance, JAX-RS API has reference implementation in the form of *Jersey*.

> In conclusion : application server is a collections of various reference implementations for the JavaEE JSRs.
>
> That is why when we code against various individuals JavaEE APIs, we can simply run it on app server; because app server bundles various JavaEE reference implementation.

- JavaEE is a JSR. For instance, JavaEE8 is a JSR366 and it's Reference implementation is Glassfish5 application server.
- JSR, or Bean Validation API, has Hibernate as its implementation.
- Java Persistence API, has EclipseLink and also hibernate as its implementation.

What is JakartaEE?

JakartaEE is essentially JavaEE going forward. It is hosted by Eclipse Foundation. JakartaEE is going to be an upgrade JavaEE.

There are a lot of members in JakartaEE project, like :

- Strategic members : Oracle, IBM, RedHat, Payara, Futisu, etc ...
- Participating members : Microsoft, Vaadin, etc ...

JavaEE/JakartaEE vs Spring Framework

In the past, JavaEE was quite complex and difficult to use, and as a result, the Spring Framework became very popular as an alternative for developing enterprise-level applications. However, one of the downsides of the Spring Framework was that developers had to write a lot of configuration files in XML, which was also not the easiest thing to write and maintain.

So, in order to make JavaEE more developer-friendly, JavaEE started to adopt some of the features of the Spring Framework.

In conclusion, JavaEE and Spring have influenced each other in terms of development practices and have evolved to provide developers with more convenient options.

Spring boot is influence by JavaEE.

> There is no JavaEE vs Spring, it is JavaEE & Spring. It is us developers that who choose what solves our problem.

- In JavaEE, it is Convention over Configuration.
- Spring also uses some of JavaEE APIs.

Done

- ☑ Day 1/90 ==> [Starting My 90-Day Journey to Learn JavaEE: Understanding the Theory and Concepts Behind JavaEE](#)

Day 2/90

Date : 03-April-2023

Setup

- Install JDK
- IDE (IntelliJ IDEA, Eclipse, NetBeans, etc)
- Install git (Version control)
- Install REST Client (Insomnia, Postman, etc) => (Intuitive REST client that makes easier to interact with RESTful endpoints easier)
- Install Maven (Download the zip and extract it in C and then add to the environment path =>
    1. Create a new *Variable* for java : JAVA_HOME, the value is JDK
    1. Create a new *Path*, and browse to the bin directory of maven)
- Install application server (Tomcat, Glassfish, **Payara**, WildFly, IBM, TomEE, etc) (I'm going to use Payara, since apache tomcat alone only supports Java Servlet & Java Server Pages (JSP) specifications, and does not have support for full JavaEE specifications & technologies like EJBs, JMS and CDI)

Deploy

Simple hello world JavaEE application :

- create a new java enterprise project (IntelliJ)
- choose JavaEE8
- add JavaEE dependency

```xml
<dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-api</artifactId>
    <version>8.0.1</version>
    <scope>provided</scope>
</dependency>
<!-- the provided scope : the container (war) will make the JavaEE APIs
available for our application, when we deploy it on app server. -->
```

- for deploying it, we need to package our application into a war file (we'll use maven to package our project) and run it on payara app server.

  Deployment : head to where the payara-micro.jar is : open a command line :

  ```
  java -jar payara-micro-6.jar --deploy
  path\to\target\folder\of_project\name_of_project.war --port 9393
  ```

  - <u>note</u> : the path to the payara-micro.jar should not have any spaces or else you'll encounter error!

Getting your feet wet | A simple training to get an overview

Trying to build a Todo application along-side the tutorial so that I can get an overview of JavaEE.

JPA : It is a set of annotation driven API that we can use to transform simple plain ol' java objects (POJOs) to entities that we can persist in Database.

- `@Id`, `@GeneratedValue(strategy = GenerationType.AUTO)`

- `@PrePersist` : we can make a method as lifecycle callback method so that if we need to initialize a property in JPA entity, it'll get executed first and initialized.

  In short : just before a property is persisted in db, the method will be set for us.

  e.g.

  ```java
  ...
  public class Todo {
      ...
      private LocalDate createdAt;

      /* to create a date on creation
       we'll create a listener or a lifecycle point in entity class to do that */
      @PrePersist
      private void init() { // to make this a lifecycle callback method :
  @PrePersist
          setDateCreated(LocalDate.now());
      }
  ```

```
        // getters & setters & constructors - or use lombok
        ...
    }
```

Done

☑ Day 2/90 ==> [Starting My 90-Day Journey to Learn JavaEE: Setup and getting Overview of JavaEE](#)

Day 3/90

Date : 04-April-2023

Continue of previous lesson (Overview of JavaEE)

Persistence Unit

Created a JPA entity. Every JPA entity needs one Persistence Unit (a collection of entities that manage together as a group).

This persistence unit will lump* all, or persist all entities as a unit that will be manage together by an entity manager.

This persistence unit is found in `src/main/resources/META-INF/persistence.xml`

- This file, configures which db it is supposed to save, update, query, and deletes the entity object.

- This file, has configuration for ORM.

it looks something like this :

```
<persistence
            xmlns="https://jakarta.ee/xml/ns/persistence"
            version="3.0">

    <persistence-unit name="todo" transaction-type="JTA">
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="javax.persistence.schema-generation.database.action"
value="drop-and-create"/>
        </properties>
    </persistence-unit>

</persistence>
```

> In order for IntelliJ to be able to create a persistence unit, we need to add JPA specification & EclipseLink as implementation in our project as dependency!
>
> Then double shift => Persistence => open persistence pallet => create a new persistence unit

```
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>eclipselink</artifactId>
      <version>4.0.1</version>
</dependency>
```

`@Transactional`

This annotation will turn a simple java class into a service.

For every method that is called, a transaction will be invoked.

How can we persist a data? we need an entity manager, it is an interface from JPA API.

```
@PersistenceContext
private EntityManager entityManager;
```

we have created an instance of entity manager (remember, this is only an overview, we'll get into these in more details later on).

`@Consumes(MediaType.APPLICATION_JSON)`

`@Produces(MediaType.APPLICATION_JSON)`

`@Inject`

`@POST, @GET, @PUT`

`@Path`

RESTful Endpoint

Creating a RESTful endpoint so that our project that we will be

using to interact with our application.

`@ApplicationPath("api/v1")` : root path to our application's endpoint. `import javax.ws.rs.ApplicationPath;`

Deployment

`mvn package` : makes a web archive (WAR) file.

Done

☑ Day 3/90 ==> My 90-Day Journey to Learn JavaEE: Day 3/90 - Continuation of getting overview on JavaEE

Day 4/90

Date : 05-April-2023

Zero progress

Today, I had **zero progress** in my learning journey. It can be discouraging, but it's important to remember that progress is not always linear. As I continue my 90-day journey of learning Java EE, Maintaining the already learned concepts of Spring Boot, and enhancing it bit by bit, and also a little bit of DSA, I am bound to have days where I don't make any progress.

Done

- ☑ Day 4/90 ==> [My 90-Day Journey to Learn JavaEE: Day 4/90 - Dealing with Zero Progress Days](#)

Day 5/90

Date : 06-April-2023

Continue of previous lesson (Overview of JavaEE)

Packaging & Deploying web application on payara micro - method 1

After our basic todo application is made, with basic CRUD operations (three architecture layer, Controller => Service => Repository => DB), we now have to deploy it on an application server.

I'm using payara server.

- We can use either the `payara micro` which is a jar file
  - for running :
    1. First build the project into war file using maven `mvn package`
    2. `java -jar payara-micro.jar --deploy location\to\the_built_war_file --port 8080`
- or use `payara full community edition`
  - for running :
    1. hello

Validation

Learned & Reviewed some bean validation annotations, such as :

- `@NotEmpty, @NotNull, @Size`

- learned new annotations :

  - `@FutureOrPresent` : user is bound to insert a date due to present or in the future, the past is not valid.

  - `@JsonDateFormat(value = "yyyy-MM-dd")` : the date that is being inserted from the REST client, will get formatted into Java understandable format.

JavaEE Uber Jar

Essentially build a fat jar with everything bundled including the application server just like spring, and we run `java -jar` to run our application or pick that jar and deploy it anywhere on JVM and it'll run.

To do so :

```
java -jar payara-micro.jar --deploy path\to\warfile.war --port 8008 --outputUberJar
any_name_for_deploying_jar.jar
```

> There is a note that I'd like to point out to...
>
> the payara micro server that I downloaded, the version is 6 which is the highest version. The problem is, this version only supports JavaEE10 & JakartaEE10. Although it will run with JavaEE/JakartaEE 8 & 9, it won't show the REST endpoints and it'd be useless for us.
>
> The version that supports JavaEE/JakartaEE 8 & 9, is payara version 5!

Packaging & Deploying with Payara micro - method 2

By using payara maven profile!

```xml
<profiles>
    <profile>
        <id>payara</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>

        <build>
            <plugins>
                <plugin>
                    <groupId>fish.payara.maven.plugin</groupId>
                    <artifactId>payara-micro-maven-plugin</artifactId>
                    <version>1.0.1</version>
                    <executions>
                        <execution>
                            <phase>package</phase>
                            <goals>
```

```
                                <goal>bundle</goal>
                            </goals>
                        </execution>
                    </executions>

                    <configuration>
                        <useUberJar>true</useUberJar>
                        <deployWar>true</deployWar>
                        <payaraVersion>5.182</payaraVersion>
                    </configuration>
                </plugin>
            </plugins>
        </build>
    </profile>
</profiles>
```

now for running : `mvn package payara-micro:start`

༼つ ◕_◕ ༽つSummarize

Java EE (now known as Jakarta EE) is indeed a collection of abstract specifications or APIs that help developers to build enterprise applications in Java. These APIs provide standardized interfaces for accessing common services such as database access, messaging, and web services.

Application servers, also known as servlet containers, are software platforms that provide an environment in which Java EE applications can run. They include a servlet engine, which handles HTTP requests and responses, and other components that provide services required by Java EE applications.

However, it's important to note that while application servers do provide an implementation of the Java EE/Jakarta EE specifications, they are not the only way to run these applications. Other deployment options include using lightweight containers such as Tomcat or Jetty, or even running Java EE applications directly on a standalone JVM without any container.

Three key APIs to JavaEE Mastery

- JPA
- CDI
- JAX-RS

> It doesn't mean you shouldn't learn other APIs!

Done

☑ Day 5/90 ==> [My 90-Day Journey to Learn JavaEE: Day 5/90 - Continuation of getting overview on JavaEE | Understanding Java EE Deployment Options and Key APIs](#)

Day 6/90

Date : 12-April-2023

CDI - Context & Dependency Injection

We need a `bean.xml` file for CDI API to get activated.

What is Dependency Injection

Dependency injection is a specific form of Inversion of control (IoC).

IoC => It is a software paradigm where individual components have there dependencies supply to them instead creating the them themselves.

So simply put, we tell the container what we want, we just declare a dependency on a specific type, and the container takes it upon itself to make that type available on the business component. We externalize the creation of object and dependencies in our application.

e.g. instead of saying `Foobar foobar = new Foobar();` we simply tell the CDI runtime that give me this particular object, and then it becomes the duty of CDI container to make that object available to you.

1. Dependency injection is a design pattern that allows components to be loosely coupled by injecting their dependencies at runtime.

2. By using a CDI container to inject dependencies, you can create more modular, testable, and maintainable code.

3. When using a CDI container, classes are loosely coupled because they do not depend **directly** on each other, but rather rely on the container to manage their dependencies.

4. Loosely coupled components make it easier to change implementations, swap out components, and unit test individual components in isolation.

5. When using dependency injection, each component can focus on doing its own job without worrying about how to create or manage other components it depends on. This means that the component's code is easier to understand and maintain, since it only needs to deal with its own logic. It also makes testing easier, because you can test each component in isolation without having to worry about the behavior of other components.

In summary, with DI we externalize the management of dependencies to the container.

Inversion of Control

Inversion of Control (IoC) is a design principle that is closely related to dependency injection. IoC refers to the idea of inverting the flow of control in a software component, where instead of the component controlling the creation and management of its dependencies, it delegates that responsibility to an external entity.

In other words, a software component should not create or manage its own dependencies; rather, it should rely on an external entity to provide them. This external entity can be a framework, a container, or any other object that manages the lifecycle of the component's dependencies.

Dependency injection is one way to achieve IoC. By using dependency injection, you are delegating the responsibility of managing the dependencies of a component to an external entity (such as a CDI container), thereby achieving IoC.

Here's an example to illustrate how IoC works:

Suppose you have a `TodoController` class that depends on a `TodoService` class to perform some business logic. Here's how you could create the `TodoService` instance using IoC:

```java
public class TodoController {
    private TodoService todoService;

    public TodoController(TodoService todoService) {
        this.todoService = todoService;
    }

    // Rest of the code ...
}
```

In this example, `TodoController` does not create or manage the `TodoService` instance. Instead, it delegates that responsibility to the caller of its constructor, which could be a framework, a test class, or any other external entity. This is an example of IoC, since `TodoController` is no longer in control of creating and managing its dependencies.

In summary, Inversion of Control is a design principle where a component's responsibility for managing its dependencies is delegated to an external entity. Dependency Injection is one way to achieve IoC by relying on an external entity (such as a CDI container) to inject dependencies into a component.

╭つ ◕_◕ ╲つSummarize

Inversion of Control (IoC) is a design principle that helps developers create more flexible software components by delegating the responsibility of managing an object's dependencies to an external entity. Instead of each component creating its own dependencies, they rely on an external entity to provide those dependencies.

Dependency Injection (DI) is a technique that implements IoC by injecting dependencies into objects instead of having the objects create or manage them themselves. This makes it easier to change implementations, swap out components, and unit test individual components in isolation.

By using DI, developers can write more modular and maintainable code with loosely coupled components that are easier to test and modify. It's an important concept to understand for anyone looking to improve their software development skills.

CDI Features

Type safe Dependency

The typesafe feature of the CDI API ensures that the dependency injection process is type-safe. This means that the compiler can detect any errors in the usage of classes, interfaces, and other types at compile time rather than at runtime.

In practical terms, this feature allows developers to use annotations to specify dependencies between components in a Java EE application. The container then automatically injects the correct dependencies at runtime, based on the information provided by the annotations.

The typesafe feature helps to reduce errors and improve maintainability by ensuring that dependencies are correctly declared and used throughout the application. It also simplifies the development process by reducing the need for manual configuration and wiring of components.

e.g.

Suppose you have a Java class that processes credit card payments. You want to ensure that this class is only injected with objects of the `CreditCardProcessor` type. Using the Typesafe feature of CDI, you can enforce this at compile time.

First, you need to define a custom qualifier annotation for the `CreditCardProcessor` class:

```java
import javax.inject.Qualifier;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Qualifier
@Retention(RUNTIME)
public @interface CreditCard {}
```

In this example, the `@CreditCard` annotation serves as a marker that you can use to identify instances of the `CreditCardProcessor` class.

Next, you need to annotate your `CreditCardProcessor` class with this new qualifier annotation:

```java
import javax.inject.Singleton;

@Singleton
@CreditCard
public class CreditCardProcessor {
    // implementation omitted for brevity
}
```

Now that your `CreditCardProcessor` class is annotated with the `@CreditCard` annotation, you can inject it into other classes using the `@Inject` annotation and the `@CreditCard` qualifier:

```
import javax.inject.Inject;

public class PaymentService {
    @Inject
    @CreditCard
    private CreditCardProcessor processor;
    // other service logic omitted for brevity
}
```

In this example, the `PaymentService` class injects an instance of the `CreditCardProcessor` class using the `@Inject` annotation and the `@CreditCard` qualifier. This ensures that only instances of the `CreditCardProcessor` class are injected, and any attempts to inject other types will result in a compile-time error.

Lifecycle Context

The lifecycle context feature of the CDI API allows for the management of the lifecycle of beans within an application.

CDI provides a set of built-in contexts, each of which is responsible for managing the lifecycle of beans in a particular way. These contexts include:

1. RequestScoped: beans that exist for the duration of a single HTTP request.

2. SessionScoped: beans that exist for the duration of a user's session with an application.

3. ApplicationScoped: beans that exist for the entire lifespan of an application.

4. Dependent: beans that are created and destroyed along with the objects that depend on them.

These different lifecycle contexts help to ensure that beans are created and destroyed at the appropriate times, based on the needs of the application. They also help to manage resource usage and prevent memory leaks by ensuring that beans are only kept in memory for as long as they are needed.

Developers can also create custom lifecycle contexts using the CDI API, allowing for even more fine-grained control over the lifecycle of beans within an application.

e.g.

Suppose you have a web application that allows users to log in and access personalized content based on their account information. You can use CDI's Context feature to scope objects to the current user session. This ensures that each user's data is kept separate from other users' data.

To do this, you might define a custom CDI scope called "SessionScoped" that corresponds to the user's session. You can then annotate your managed beans with this scope to ensure that they are only available within the scope of the current user's session.

Here's some example code to illustrate how this might work:

```
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;

`@SessionScoped`
public class UserAccount implements Serializable {
    private String username;
    private String password;
    // getters and setters omitted for brevity
}
```

In this example, the `UserAccount` class is annotated with the `@SessionScoped` annotation, which tells CDI to create a new instance of this class for each user session. The `Serializable` interface is included so that instances of this class can be stored in the user session.

You can then inject instances of this class into other managed beans using the `@Inject` annotation:

```
import javax.inject.Inject;

public class UserProfileController {
    @Inject
    private UserAccount userAccount;
    // other controller logic omitted for brevity
}
```

In this example, the `UserProfileController` class injects an instance of the `UserAccount` class using the `@Inject` annotation. Because the `UserAccount` class is annotated with `@SessionScoped`, a new instance of this class will be created for each user session, ensuring that each user's data is kept separate.

Interceptors

Just as the name implies, The Interceptor feature in CDI API allows you to define interceptors for your managed beans. An interceptor is a class that can intercept method invocations on another class, allowing you to add cross-cutting concerns such as logging or security checks. they intercept the requests to methods so you can have interceptors do cross-cutting work for your business application.

For instance, I can have a method that before it gets invoked, I want to log certain specifics properties of the request or whoever is logged-in, I use interceptor to do that. So based on our logic implementation, after the interceptor is invoked before the method, we can then allow it to proceed or abort the request.

e.g.

Suppose you have a method that performs some expensive computation, and you want to log how long it takes to run. You can use an interceptor to log the method's execution time without modifying the original method.

First, you need to define an interceptor class with a method that logs the method's execution time:

```
import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

@Interceptor
public class PerformanceLoggingInterceptor {
    @AroundInvoke
    public Object logPerformance(InvocationContext context) throws Exception {
        long startTime = System.currentTimeMillis();
        try {
            return context.proceed();
        } finally {
            long endTime = System.currentTimeMillis();
            System.out.println("Method " + context.getMethod().getName() + " took "
+ (endTime - startTime) + "ms to execute.");
        }
    }
}
```

In this example, the `PerformanceLoggingInterceptor` class is defined as an interceptor by annotating it with the `@Interceptor` annotation. The `logPerformance` method is annotated with the `@AroundInvoke` annotation, which indicates that it should be invoked before and after the intercepted method call. Within this method, we record the start time, call the actual method using `context.proceed()`, record the end time, and print a message indicating how long the method took to execute.

Next, you need to annotate your target method with the `@Interceptors` annotation to apply the interceptor:

```
import javax.ejb.Stateless;
import javax.inject.Inject;
import javax.interceptor.Interceptors;

@Stateless
public class MyService {
    @Inject
    private SomeDependency dependency;

                @Interceptors(PerformanceLoggingInterceptor.class)
    public void doSomethingExpensive() {
        // Expensive computation
        dependency.doSomethingElse();
    }
}
```

In this example, the `doSomethingExpensive` method is annotated with the `@Interceptors` annotation and passed in the `PerformanceLoggingInterceptor` class. This indicates that any calls to the `doSomethingExpensive` method should be intercepted by the `PerformanceLoggingInterceptor`.

When you run your application and call the `doSomethingExpensive` method, the `PerformanceLoggingInterceptor` will intercept the method call and log how long it took to execute.

By using an interceptor, you can add additional functionality or behavior to a method without modifying the original method, making your code more maintainable and flexible.

Events

The Event feature in CDI API provides a way to decouple components in an application by allowing them to send and receive messages asynchronously.

The Event feature is based on the Observer pattern, where one component (the observer) registers to receive notifications from another component (the subject). In CDI, the subject is called the "event producer" and the observer is called the "event consumer".

It's a way for us to develop a highly decoupled applications such that, one component can send data to another component without any form of connection or relation between them.

You can create an event, then you can fire that event. Once you have done that, you have listeners to listen in for firing the events. Those listeners will be informed of the event.

CDI 2.0 introduced Asynchronous event.

e.g.

Let's say you have an application that manages a list of tasks. Whenever a task is completed, you want to log a message to the console saying that it has been completed.

First, you define an event type that represents a completed task:

```java
public class TaskCompletedEvent {
    private final String taskId;

    public TaskCompletedEvent(String taskId) {
        this.taskId = taskId;
    }

    public String getTaskId() {
        return taskId;
    }
}
```

Next, you create a class that will produce events when tasks are completed:

```java
@ApplicationScoped
public class TaskManager {
    @Inject
    private Event<TaskCompletedEvent> taskCompletedEvent;

    public void completeTask(String taskId) {
        // Do some logic to mark the task as completed

        // Fire a TaskCompletedEvent
        taskCompletedEvent.fire(new TaskCompletedEvent(taskId));
    }
}
```

This class has a method called `completeTask` which takes a `taskId` and performs some logic to mark the task as completed. After that, it creates a new `TaskCompletedEvent` object with the `taskId` and fires it using the `taskCompletedEvent` instance.

Finally, you create a class that observes the `TaskCompletedEvent` and logs a message to the console:

```java
@RequestScoped
public class TaskLogger {
    public void logTaskCompletion(@Observes TaskCompletedEvent event) {
        System.out.println("Task " + event.getTaskId() + " has been completed.");
    }
}
```

This class has a method called `logTaskCompletion` which observes the `TaskCompletedEvent`. When an event is fired, this method is called and logs a message to the console indicating that the task has been completed.

Now, whenever you call the `completeTask` method on the `TaskManager`, an event will be fired and the `TaskLogger` will log a message to the console indicating that the task has been completed.

Service Provider Interface (SPI)

The SPI (Service Provider Interface) feature in CDI (Contexts and Dependency Injection) API allows third-party providers to extend or replace the default behavior of the CDI container. This is achieved by implementing specific interfaces defined in the CDI specification, which allows the provider to provide its own implementations of key components such as bean discovery, injection resolution, and context management.

Using the SPI feature, custom extensions can be developed for specific use cases that are not covered by the standard CDI functionality. For example, a provider might implement a custom bean discovery mechanism to automatically discover beans in a specific package or provide an alternative context implementation for managing scoped objects. It it used to make our own CDI API extensions. Just like we have finder queries in spring data JPA, we also have the same thing in JavaEE platform through the use of SPI interface with a CDI extension.

e.g.

Let's say you have an application that uses CDI for dependency injection, and you want to provide a custom implementation for handling transactions. You can use the SPI feature to extend the CDI container with your own transaction management implementation.

First, you need to create a provider class that implements the `javax.enterprise.inject.spi.Extension` interface. This class will define the behavior of your custom extension:

```java
public class TransactionExtension implements Extension {
    public void beforeBeanDiscovery(@Observes BeforeBeanDiscovery event) {
        // register our custom Bean<TxManager> with the container
        event.addAnnotatedType(TxManager.class, "txManager");
    }

    public void processInjectionTarget(@Observes ProcessInjectionTarget<?> event) {
        AnnotatedType<?> annotatedType = event.getAnnotatedType();
        if (annotatedType.getJavaClass().equals(MyService.class)) {
            // replace the default TxManager injection with our custom one
            InjectionTarget<?> target = event.getInjectionTarget();
            event.setInjectionTarget(new MyServiceInjectionTarget(target));
        }
    }
}
```

In this example, we're defining a custom `TxManager` bean and replacing the default injection of `TxManager` in `MyService` with our custom implementation.

Next, you need to specify the provider class in a file named `javax.enterprise.inject.spi.Extension` located in the `META-INF/services` directory of your application:

```
com.example.TransactionExtension
```

Finally, you can inject your custom `TxManager` bean into your service using the `@Inject` annotation as follows:

```java
public class MyService {
    @Inject
    private TxManager txManager;

    // ...
}
```

By doing this, the CDI container will invoke the `TransactionExtension` methods to register the custom `TxManager` bean and replace the default injection of `TxManager` in `MyService` with our custom implementation.

Bean Discovery Mode

Bean discovery mode is a feature in CDI that determines how beans are discovered and registered by the container. There are two bean discovery modes in CDI:

1. Annotated Bean Discovery Mode: In this mode, the CDI container discovers beans based on annotations such as `@javax.inject.Named`, `@javax.enterprise.context.RequestScoped`, `@javax.enterprise.inject.Produces`, etc. Any Java class with one or more of these annotations is considered a bean and is automatically registered with the container.

2. All Bean Discovery Mode: In this mode, the CDI container discovers all Java classes in the archive (classpath) and registers them as beans, unless they are explicitly excluded using an extension or other configuration mechanism.

The default bean discovery mode in CDI 2.0 is annotated discovery mode. However, you can change the bean discovery mode using the beans.xml file, which is a deployment descriptor for CDI archives.

my own words : It refers to a mechanism which the DI runtime, analyzes and then discovers beans for it.

We need a `bean.xml` file for CDI API to get activated. And if by default CDI API is omitted, then we get the `annotated` mode.

`Annotated` : Beans that will be eligible by CDI runtime, are classes that are annotated with certain specific CDI annotations.

So, bean discovery means, that the CDI will scan your archive at boot time, and then will gather all those beans that are annotated with CDI specific annotations and will make them eligible for management.

`All` : Every single bean that we create in our application, is eligible to be managed by JavaEE CDI runtime.

CDI Container

In the context of the CDI API, a container refers to the runtime environment that manages the lifecycle of objects and their dependencies. It's responsible for discovering beans (managed objects), instantiating them, and injecting their dependencies.

The CDI container provides a set of services for managing object lifecycles, such as defining scopes for beans, managing injection points, and handling events. It also provides a set of built-in contexts that define the lifecycle of a bean, such as request, session, and application scopes.

In summary, the CDI container is the central component that manages the lifecycle of objects in a CDI-enabled application.

> A container is like a factory, where Java classes goes in, and comes out with certain specific features and functionality.

Done

☑ Day 6/90 ==> JavaEE: Day 6/90 - Context and Dependency Injection (CDI)

Day 7/90

Date : 13-April-2023

CDI - Context & Dependency Injection

What are Beans and Contextual Instances?

Bean : a bean is simply a template that a developer makes.

Contextual Instance : it is an instance of a bean that is created by CDI container and managed by it.

CDI Injection Point

what is an injection point? CDI Injection Point is one of the core concepts of CDI. It is the point where the CDI container can inject the dependency for you.

An Injection point is a location in your code where you want to inject a particular object or value to satisfy a dependency. In CDI, Injection points are represented by specific annotations such as `@Inject`.

The CDI container uses these Injection Points to resolve dependencies for you automatically at runtime, without you having to explicitly provide objects or values. The container looks for objects that match the type of the Injection Point, and it provides the matching object to the Injection Point.

=> So, a CDI Injection Point is simply a place in your code where you want to use automatic dependency injection provided by the CDI container. You can annotate a field, method, or constructor with the `@Inject` annotation to mark it as an Injection Point.

Field, Constructor and method injections are the Injection Points!

> If we don't inject into a dependency, and we go ahead and use it in our methods, we'll get `NullPointerException` because we nor the CDI container, nor ourselves created any instance of that dependency.

Field Injection

Field Injection is a type of dependency injection where dependencies are injected into the fields of a class. In Java, this is typically done using the `@Inject` annotation from the CDI framework.

> It requests the CDI container for a contextual instance, to be injected into a particular field.

```java
public class MyService {
    @Inject
    private MyDependency myDependency;
    // ...
}
```

One advantage of Field Injection is that it can make **your code more concise**, since you don't need to create constructor or setter methods just to inject dependencies. However, some argue that it can make your code less testable, since it can be harder to mock dependencies for unit testing.

Constructor Injection

Constructor Injection is a type of dependency injection where dependencies are injected via a class constructor. In Java, this is typically done using the `@Inject` annotation from the CDI framework.

```java
public class MyService {
    private final MyDependency myDependency;

    @Inject
    public MyService(MyDependency myDependency) {
        this.myDependency = myDependency;
    }
    // ...
}
```

One advantage of Constructor Injection is that it can make your code more **testable**, since it allows you to easily inject mock dependencies for unit testing. It also helps ensure that all required dependencies are available before a new instance of the class is created.

Method Injection

In the context of the CDI API in Java EE, method injection refers to a way of injecting dependencies directly into a method of a bean instead of injecting them through the constructor or setter methods.

```java
public class MyBean {

  private MyDependency dependency;

  @Inject
  public void setDependency(MyDependency dependency) {
    this.dependency = dependency;
  }
  //...
}
```

Method injection can be useful when you need to inject dependencies into a specific method of a bean, rather than to the bean's constructor or setter methods. It can also be used to inject dependencies into non-public methods, which cannot be done with constructor or setter injection.

CDI Lifecycle Callback

In the CDI API, a lifecycle callback is a method that gets invoked by the container at various points during the lifecycle of a bean.

There are two types of lifecycle callbacks in CDI:

1. Initialization callbacks: These methods are called after dependency injection has occurred but before the bean is put into service. They are annotated with the `@PostConstruct` annotation.

2. Destruction callbacks: These methods are called when the bean is being destroyed or removed from service. They are annotated with the `@PreDestroy` annotation.

Initialization callbacks are useful for performing any initialization work that needs to be done before the bean can be used. For example, initializing a database connection or setting up a logger.

Destruction callbacks are useful for releasing any resources that the bean has acquired during its lifetime. For example, closing a database connection or releasing a file handle.

in my own words : Lifecycle callback, is a point in a lifecycle of a bean, that the CDI container gives us the opportunity to do certain specific things.

PostConstruct

In the CDI API, `@PostConstruct` is a lifecycle callback method that is invoked immediately after a managed bean has been instantiated and its dependencies have been injected.

The `@PostConstruct` annotation can be applied to any method of a managed bean class, and that method will be called automatically by the CDI container after the object has been constructed and all its dependencies have been injected.

Typically, you would use `@PostConstruct` to perform any initialization or setup that needs to happen after the bean has been created but before it is used. For example, you might use `@PostConstruct` to open database connections, initialize data structures, or start background threads.

By using lifecycle callback methods like `@PostConstruct`, you can separate the logic for constructing and initializing an object from the rest of the application logic, which can make your code more modular, easier to test, and easier to maintain.

> It is a point at which all the beans & dependencies have been created and all initialization have been completed, and are ready to use just before putting it in action or putting it in service!

PreDestroy

In the CDI API, `@PreDestroy` is a lifecycle callback method that is invoked just before a managed bean is destroyed by the container.

The `@PreDestroy` annotation can be applied to any method of a managed bean class, and that method will be called automatically by the CDI container just before the object is destroyed. Typically, you would use `@PreDestroy` to perform any cleanup or teardown that needs to happen before the bean is destroyed. For example, you might use `@PreDestroy` to close database connections, release resources, or stop background threads.

By using lifecycle callback methods like `@PreDestroy`, you can ensure that any necessary cleanup happens in a timely and orderly manner, without relying on the garbage collector to handle it for you. This can help prevent resource leaks and other issues that can occur if you don't properly clean up after your objects.

It's important to note that the exact timing of `@PreDestroy` method invocation is not guaranteed. The CDI specification only requires that `@PreDestroy` methods are called before the bean is destroyed, but it does not specify when exactly that will happen.

> It gets invoked just before the bean & or dependency is destroyed and made available for garbage collection.

Managed Beans & Bean Types

Managed Bean : the CDI API, a managed bean is a Java object that is instantiated, initialized, and managed by the CDI container. Managed beans are used to implement the business logic and control flow of an application.

Managed beans are annotated with the `@javax.inject.Named` annotation or the `@javax.enterprise.context` annotations such as `@RequestScoped`, `@SessionScoped`, `@ApplicationScoped`, etc. These annotations define the scope of the bean and its lifecycle within the container.

Bean Types : In the CDI (Contexts and Dependency Injection) API, a bean type is a type that may be injected or looked up by its clients. A bean type can be a class or an interface, and it is used to define the contract between the producer of a bean and its consumer.

A bean type must be specified on the `@javax.enterprise.inject.Produces` annotation, which is used to declare a producer method or field. The producer method or field must return an object whose class or interface matches the bean type.

Bean types are also used in qualifiers, which are annotations that further specify the injection point of a bean. Qualifiers allow you to differentiate between multiple beans of the same type that have different characteristics or configurations.

> What is a Managed Bean?
>
> A managed bean, is any bean that 1. it is eligible for CDI management/injection, 2. it is managed by CDI container.

> What is a Bean Type?
>
> Bean type refers to a concrete type of a bean, or the type to which a bean is related, such that we can say "this bean, is of this type".

CDI Qualifiers

In the CDI API, a qualifier is a type-safe way to distinguish between beans *that implement the same interface or extend the same class*. Qualifiers allow you to specify which bean to use when there are multiple beans of the same type in the application context.

A qualifier is defined as an annotation that is applied to a bean, and it can include additional metadata that helps to further differentiate the bean. For example, the `@Named` annotation is a built-in qualifier in CDI that allows you to give a bean a unique name.

Here's an example of how you might use a custom qualifier annotation:

```java
@Qualifier
@Retention(RUNTIME)
@Target({ ElementType.TYPE, ElementType.METHOD, ElementType.FIELD,
ElementType.PARAMETER })
public @interface MyQualifier {
    String value();
}
```

With this custom qualifier annotation, you can annotate your beans like this:

```java
@MyQualifier("foo")
public class FooBean implements MyInterface { ... }

@MyQualifier("bar")
public class BarBean implements MyInterface { ... }
```

Then, in another bean where you want to inject one of these two beans, you can specify which one to use based on the qualifier:

```java
@Inject
@MyQualifier("foo")
private MyInterface myFoo;

@Inject
@MyQualifier("bar")
private MyInterface myBar;
```

This tells CDI to inject the bean with the `@MyQualifier("foo")` annotation into the `myFoo` field, and the bean with the `@MyQualifier("bar")` annotation into the `myBar` field.

another example :

Suppose we have a `Salute` interface with one method. This interface is implemented by 2 Java classes (`Police`, `Soldier`). When we inject the implementation of this interface, the CDI container won't know which are you calling. Do you mean salute of police? or the soldier? so there is an ambiguity. To resolve this, we'll use qualifier, and mark our implementation classes with them to separate them and make them distinguishable for CDI container.

```java
public inteface Salute {
    String salute(String salute);
}
```

java classes that implement salute :

```
@Police
// other annotations...
public class Police implements Salute {
    @Override
    public String salute(String salute) {
        return MessageFormat.format("Sir, Yes Sir, {0}", salute);
    }
}
```

```
@Soldier
// other annotations...
public class Soldier implements Salute {
    @Override
    public String salute(String salute) {
        return MessageFormat.format("All Hail to, {0}", salute);
    }
}
```

```
public Class DemoQualifierBean {
    @Inject
    @Police
    private Salute policeSalute;

    @Inject
    @Soldier
    private Salute soldierSalute;

    // other methods that use the Salute ...
}
```

Now the CDI container knows, when we call the police salute, we mean Police's implementation and same for soldier.

> Qualifiers are annotations that you create/use, to tell the CDI container the exact type of contextual instance (instance of bean/dependency) to be resolved to.

Creating Qualifiers with Values

From the previous example, Instead of creating 2 different qualifier interfaces, we can make one qualifier that takes values of a specific type.

```java
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.TYPE, ElementType.METHOD,
ElementType.PARAMETER})
public @interface ServiceMan {
    ServiceType value();

    public enum ServiceType {
        SOLDIER, POLICE
    }
}
```

the changes for the `Soldier` and `Police` :

```java
//  @Soldier   <-- we won't use single qualifier
@ServiceMan(value = ServiceMan.ServiceType.POLICE)
public class Soldier implements Salute {
    @Override
    public String salute(String salute) {
        return MessageFormat.format("All Hail to, {0}", salute);
    }
}
```

```java
// @Police   <-- we won't use single qualifier
@ServiceMan(value = ServiceMan.ServiceType.POLICE)
public class Police implements Salute {
    @Override
    public String salute(String salute) {
        return MessageFormat.format("Sir, Yes Sir, {0}", salute);
    }
}
```

the same goes for the class that is calling :

```java
public Class DemoQualifierBean {
    @Inject
    @ServiceMan(value = ServiceMan.ServiceType.POLICE)
//    @Police   <-- not using single interface qualifier
    private Salute policeSalute;

    @Inject
    @ServiceMan(value = ServiceMan.ServiceType.POLICE)
//    @Soldier  <-- not using single interface qualifier
    private Salute soldierSalute;

    // other methods that use the Salute ...
}
```

more efficient ;)

CDI Stereotypes

In CDI (Contexts and Dependency Injection), a stereotype is a specialized annotation that allows developers to quickly apply a set of related annotations to a class.

The CDI API provides several built-in stereotypes, such as `@Model`, `@Controller`, and `@Repository`, which are commonly used in web application development with the Model-View-Controller (MVC) design pattern.

By using a stereotype annotation, you can apply a group of related annotations to a class with a single annotation, rather than individually annotating each field or method. This can help make your code more concise and easier to read.

example for creating our own stereotype :

```
@Stereotype
@RequestScoped
@Named
@Retention(RetentionPolicy.RUNITME)
@Target(ElementType.TYPE) //class level only
public @interface Web {


}

// now we can use @Web anywhere we want
```

> what are stereotypes?
>
> There are times where we need to use different annoations on single type, and repeating it throughout is very tedious and repetitive.
>
> Stereotype is collection of annotations grouped together as one, to solve this tedious problem.
>
> => It is same as `@RestController` in spring boot which is a combination of `@Controller` & `@ResponseBody`.

@Named

The `@Named` annotation is a Java annotation that can be used to specify a name for a bean or resource in a Java EE application. When using dependency injection, the `@Named` annotation can be applied to a class, allowing it to be referred to by name in other parts of the application. The `@Named` annotation is primarily used for naming beans or resources in Java EE applications, and making them available for injection using dependency injection frameworks like CDI.

While it is true that the `@Named` annotation is commonly used in conjunction with JavaServer Faces (JSF) to make beans accessible from web pages, this is achieved through the use of EL (Expression Language), rather than by exposing public properties directly.

CDI Scopes & Contexts

Scopes define the lifecycle of a managed bean or a contextual instance. A scope defines the context in which a bean instance exists, and thus determines how long an instance will be preserved and when it will be destroyed.

Contexts in CDI refer to the runtime environment that manages the lifecycle of a bean instance. A context is responsible for creating and destroying bean instances, as well as managing their state. Each scope defines a separate context.

There are several built-in scopes in CDI, including:

1. `@ApplicationScoped` : The bean instance is created once for the entire application and lives until the application shuts down.

2. `@SessionScoped` : The bean instance is created once per user session and lives until the session ends.

3. `@RequestScoped` : The bean instance is created once per HTTP request and lives until the request is completed.

4. `@Dependent` : This is the default scope if no other scope is specified. The bean instance is dependent on the lifecycle of its injection point and is destroyed when its injection point is destroyed.

> What is a scope?

A scope is simply a way to tell the container to associate a specific contextual instance (instance of dependency) with a given context.

A real world analogy to understand more properly :

Let's say you're running a coffee shop and you have different types of customers who visit your store:

Regular customers: They visit your coffee shop frequently and are loyal customers.

Occasional customers: They visit your coffee shop once in a while.

One-time customers: They visit your coffee shop just once.

Now, let's see how CDI scopes can be related to these customer types:

1. `@ApplicationScoped` : This is like the regular customers. The same instance of a bean is maintained throughout the lifetime of the application, just like how regular customers keep coming back to your coffee shop.

2. `@SessionScoped` : This is like occasional customers. A new instance of a bean is created when a user/session starts and is maintained throughout the session, just like how occasional customers come to your coffee shop once in a while and stay for a specific period of time.

3. `@RequestScoped` : This is like one-time customers. A new instance of a bean is created for each request made to your server, just like how a one-time customer makes only one purchase at your coffee shop and then leaves.

4. `@Dependent` : This is like customers who borrow things. A new instance of a bean is created anytime an object needs it, and is destroyed when the object no longer needs it. It is like borrowing something from someone, you use it for as long as you need it and then return it back.

> What is a context?

A context refers to, a valid environment where a contextual instance can reside.

Let me try to explain the concept of context in CDI with a real-world analogy :

Imagine you are throwing a party and you have different rooms with different themes:

1. The dance floor: This is where people come to dance and have fun.

2. The bar: This is where people come to get drinks and socialize.

3. The lounge: This is where people come to relax and chat.

Now, let's see how contexts are related to these party rooms:

A context defines the runtime environment that manages the lifecycle of a bean instance. In our analogy, a context would be like the environment within each room that determines how long people stay and what they do while they're there.

For example:

1. The dance floor context: This context manages the lifecycle of bean instances related to dancing and having fun. People come here to dance, and the context ensures that the music keeps playing and the dance floor stays active as long as people want to stay and dance.

2. The bar context: This context manages the lifecycle of bean instances related to drinks and socializing. People come here to get drinks and chat with friends, and the context ensures that there are always bartenders available to serve drinks and create a welcoming atmosphere for socializing.

3. The lounge context: This context manages the lifecycle of bean instances related to relaxation and conversation. People come here to sit down and chat with friends, and the context ensures that the space is comfortable and conducive to conversation.

In CDI, contexts provide a way to manage the lifecycle of beans and ensure that they exist only for as long as they are needed. Just like the context of each party room manages the environment within that room, CDI contexts manage the environment within which bean instances exist.

@Dependent

`@Dependent` scope is a built-in bean scope that indicates that an instance of a bean has a lifecycle **that is bound** to the lifecycle of **its injection point**.

In other words, when you inject a dependent-scoped bean into another bean, the container will create a new instance of the dependent-scoped bean for each injection point. This means that the dependent-scoped bean instances are not shared between injection points.

One way to think about the `@Dependent` scope is to compare it to a disposable coffee cup.

Imagine you're at a café and you order a coffee. The barista hands you a disposable cup with your coffee in it. This cup is dependent on your coffee order - it was created specifically for you, and it will be discarded once you've finished your drink.

Now imagine that you order another coffee, and the barista hands you another disposable cup. This cup is also dependent on your coffee order - it's a new cup created specifically for this new order. It's not the same cup as before, and it's not shared with anyone else.

In a similar way, when you inject a bean with a `@Dependent` scope into another bean, the container creates a new instance of that bean specifically for that injection point. That instance is not shared with any other injection points, and it will be discarded once the injection point is destroyed.

The `@Dependent` scope is the default scope for a bean if no other scope is specified. It is also sometimes referred to as the "pseudo-scope" because it has no real scope and is essentially the absence of any explicit scope annotation.

@RequestScope

The `@RequestScope` annotation is used to define a bean's scope to be scoped to an HTTP request.

When you annotate a bean with `@RequestScope`, Spring creates a new instance of that bean for every HTTP request that comes into your application. This means that each user request will get its own unique instance of the bean.

This can be useful when you have objects that store information related to a specific user request, such as data entered on a form or the user's selected language preference. By using request-scoped beans, you ensure that each user request gets its own separate instance of these objects, preventing any interference or confusion between concurrent requests.

> we tell the CDI container, that the contextual instances of bean, to be associated with a `@RequestScope` i.e. an HTTP request.

@SessionScoped

`@SessionScoped` annotation defines a bean's scope as "session". This means that a single instance of this bean will be created for each user session, and it will be available for the entire duration of that session.

This is useful when you need to maintain stateful information across multiple HTTP requests made by the same user. For example, if you have a shopping cart feature on your website, you could store the contents of the cart in a `@SessionScoped` bean so that the user's cart is persisted between page loads.

It's worth noting that the `@SessionScoped` annotation requires a mechanism for storing session data, such as cookies or URL rewriting. The CDI API doesn't provide this functionality, so you'll need to use a compatible web framework that includes session management to fully leverage the benefits of `@SessionScoped` beans.

@ApplicationScoped

In the CDI (Contexts and Dependency Injection) API, `@ApplicationScoped` is a built-in scope annotation that specifies that a bean's context is tied to the lifecycle of the application. This means that there will be only one instance of the bean created for the entire duration of the application.

The `@ApplicationScoped` annotation is used to indicate that an object should be instantiated once per application and shared across all requests and sessions. This can be useful for objects that need to be shared across multiple users or requests, such as application configuration objects or database connection pools.

When a bean is annotated with `@ApplicationScoped`, it is instantiated when the application starts up and destroyed when the application shuts down. Any state stored in the bean is available to all parts of the application and can be safely accessed from multiple threads simultaneously.

> It is a class level annotation. It is used to create single instances of a particular bean.
>
> Let me say it again : Applicaiton scoped, will create a single contextual instance of a bean type and associate with the lifetime of the applciation itself.
>
> It is basically a singleton that last throghout the lifetime of the application, and the container is responsible for managing that bean for us.

@ConversationScoped

In the CDI (Contexts and Dependency Injection) API, `@ConversationScoped` is a built-in scope annotation that specifies that a bean's context is tied to a specific user conversation. This means that there will be only one instance of the bean per conversation.

The `@ConversationScoped` annotation is used to indicate that an object should be instantiated once for each user session and shared across multiple requests within that session. This can be useful for objects that need to maintain state across multiple requests from the same user, such as shopping carts or wizards.

When a bean is annotated with `@ConversationScoped`, it is instantiated when a new conversation is started and destroyed when the conversation ends. A conversation is usually started by the user performing some action that requires interaction over multiple pages or views.

To use the `@ConversationScoped` annotation, you first need to start a conversation by invoking the `begin()` method on an instance of the `javax.enterprise.context.Conversation` interface. You can then inject the conversation into your bean using the `@Inject` annotation.

> It is genarally used for Java Server Faces (JSF) APIs.
>
> It is bound to a context that is similar to `@SessionScoped`, but then it is manually managed by developer.

Done

Day 8/90

Date : 14-April-2023

CDI - Context & Dependency Injection

Context & Scopes in action

`@RequestScoped` : Every time an HTTP request is called, container should cause creation of a new bean.

> So we expect to see a **new** hashcode every time this bean is created.

```
@RequstScoped
public class RequestScope {
    public String getHashCode() {
        return this.hashCode() + " ";
    }
}
```

`@SessionScoped` : 1) This bean is bound to an HTTP session. 2) Also, if we open the request in another browser for instance, a new session will be created.

> 1. So we expect to see a **single** hashcode repeated for a given session.
>
> 2. So we expect to see **another** hashcode repeated for that session.
>
> > Session scope, manages the bean per client!

```
@SessionScoped
public class SessionScope {
    public String getHashCode() {
        return this.hashCode() + " ";
    }
}
```

`@ApplicationScoped` : It is singleton and only one contextual instance is created throughout the lifetime of application.

> So we expect to see the **same** hashcode, despite making a new request in new browser, tab, etc
> ...

```
@ApplicationScoped
public class ApplicationScope {
    public String getHashCode() {
        return this.hashCode() + " ";
    }
}
```

`@DependentScoped` : Since it is dependent, wherever we inject it, it should inherit that context.

```
public class DependentScope {
    public String getHashCode() {
        return this.hashCode() + " ";
    }
}
```

_____Now Test It_____

```
// annotations ...
public class ScopesBean {
    @Inject
    private RequestScope requestScope;

    @Inject
    private SessionScope sessionScope;

    @Inject
    private ApplicationScope applicationScope;

    @Inject
    private DependentScope dependentScope;

    // other methods hidden for brevity ...
    // getters setter ...
}
```

⌐つ◕_◕ \つSummarize

Contexts in CDI refer to a set of related objects that share a lifecycle and are managed by the container. *Scopes*, on the other hand, define the lifecycle of a bean instance within a particular context. It is basically a way to associate (manage) a bean with a given context.

There are several built-in scopes in CDI:

1. `@ApplicationScoped` - Beans with this annotation have a lifecycle that is tied to the application itself. They are created when the application starts up and destroyed when the application shuts down.

2. `@SessionScoped` - Beans with this annotation have a lifecycle that is tied to a user session. They are created when a user session is established and destroyed when the session ends.

3. `@ConversationScoped` - Beans with this annotation have a lifecycle that is tied to a specific conversation between the user and the application. They are created when the conversation starts and destroyed when the conversation ends.

4. `@RequestScoped` - Beans with this annotation have a lifecycle that is tied to a single HTTP request. They are created when the request is received and destroyed when the response is sent.

5. `@Dependent` - Beans with this annotation have a lifecycle that is tied to the lifecycle of the object that injects them. They are created when the injecting object is created and destroyed when it is destroyed.

CDI Producers

In CDI, a producer method is a method that creates and returns a bean instance for injection. It allows you to customize how a bean instance is created, including how its dependencies are injected.

To define a producer method, you first annotate it with the `@Produces` annotation. You then declare the bean type of the produced instance using the return type of the method. You can also use additional annotations to specify the scope of the produced instance or qualifier annotations to further identify the produced instance.

For example, suppose you have a `Logger` interface and want to inject an implementation of this interface into your application. You could define a producer method like this:

```java
import javax.enterprise.inject.Produces;
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class LoggerFactory {
    @Produces
    public Logger createLogger() {
        return new ConsoleLogger();
    }
}
```

This producer method creates an instance of `ConsoleLogger` and makes it available for injection wherever an instance of `Logger` is required. The `@ApplicationScoped` annotation on the `LoggerFactory` class specifies the scope of the produced instance.

> What is a CDI producer?
>
> It is an API construct, that allows developers, to tend classes that we don't own into CDI managed and injectable beans. Whatever the method returns, it is eligible for CDI injection.
>
> - The producer method **MUST** have a return type and should not be of type void.
> - If the method marked with `@Produces` has parameters, the property inside the parameter **MUST** be an injectable bean.

Scoping Returned Beans

what will be the scope of the producer methods? It will be `@Dependent` .

Field Producers

same as method producer, it just depends on your use case.

- If you need to do some kind of work in the method using let's say the parameters, use method producer.

- an example :

```
@Produces
@PersisteceContext
EntityManager entityManager;

// now whereever we want to inject
// ...
@Inject
EntityManager manager;
```

Qualifying Beans

How do we clarify an ambiguity to the CDI runtime?

We use CDI qualifiers to avoid ambiguity.

in our previous example "CDI Qualifiers - Day 7",

```
// ...
@Produces
@Police // this will solve the ambiguity
public Salute getSalutation() {
    return new Police();
}
```

Disposers

Let's explain this with example :

```
// ...
@Produces
public List<String> getLuckyDish() {
    List<String> dishes = List.of("food1","food2","food3");
    return dishes;
}


public void dispose(@Disposes List<String> dishes) {
    dishes = null;
}
```

Disposers basically inverse of producers. It gives the chance to custom cleaner. It disposes of the bean created.

╭つ ●_●╲つSummarize

CDI *Producer*

- is a way to transform mostly classes that we don't own into CDI managed beans.

- We can use either producer <u>method</u> or <u>field</u>. whatever is returned is eligible for injection.

- We can also Scope the method or field.

- We can also prevent ambiguity by using qualifiers.

CDI *Disposer*

- is a void method that has parameters with `@Disposes` and the method will inject the custom producer and you can do custom cleanup on it in you disposer method.

Done

☑ Day 8/90 ==> [JavaEE: Day 8/90 - Context and Dependency Injection (CDI)](#)

Day 9/90

Date : 15-April-2023

CDI - Context & Dependency Injection

CDI Interceptors

In the context of the CDI API, interceptors are a type of component that can intercept method invocations and perform additional operations before or after the method is called.

CDI interceptors are defined using an annotation, `@Interceptor`, and can be used to implement cross-cutting concerns such as logging, security, or performance monitoring across multiple beans in an application.

To define an interceptor, you would create a class and annotate it with `@Interceptor`. Within the class, you can define methods that intercept calls to other classes and execute additional logic.

CDI interceptors operate using a chain-of-responsibility pattern. When a method is invoked on an intercepted bean, the interceptor chain is executed, with each interceptor potentially modifying the behavior of the method before passing control to the next interceptor in the chain. The final interceptor in the chain invokes the original method.

To declare interceptors in JavaEE, we need to do 2 things :

1. We need to declare an `@InterceptorBinding` (an interface)

```
// example
@InterceptorBinding
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
@Inherited
public @interface Logged {
}
```

`@InterceptorBinding` : is an annotation that we'll be using to trigger an interceptor or to trigger a method to be intercepted or a class to be intercepted.

2. After that, we need to declare an interceptor binding code. We can think of it as the implementation of it. This interceptor binding code will run, when a method or class is intercepted.

```
// from previous example
// Bind interceptor to this class
@Logged // the bining interface we declared earlier
@Interceptor
@Priority(Interceptor.Priority.APPLICATION)
public class LoggedInterceptor {
    @Inject
    private Logger logger;

    // mocked user; could be from db
    private String user = "user";

    @AroundInvoke
    public Object logMethodCall(InvocationContext context) throws Exception {
        // for example, log user who called method and time
        logger.log(Level.INFO, "User {0} invoked {1} method at {2}", new
Object[]{user, context.getMethod().getName(), LocalDateTime.now()});
        return context.proceed();
    }
}
```

`@AroundInvoke` : The `@AroundInvoke` annotation indicates that the `logMethodCall` method intercepts all method invocations.

`InvocationContext` : The `InvocationContext` parameter provides access to information about the intercepted method, such as its name and parameters.

> Exposes contextual information about the intercepted invocation and operations that enable interceptor methods to control the behaviorof the invocation chain.
>
> ~ Java Documentation

==> So now wherever we want to intercept a method, in this example, we'll annotate that method with `@Logged` annotation. Anytime the method is invoked, the container will come to the `@AroundInvoke` marked method and execute what's in the method. After whatever logic it has done, It will then proceed to the target method (`return context.proceed()`).

> Interceptors are similar to Aspect-Oriented-Programming (AOP). It is a way for us to intercept calls to a method or entire methods in a given class.
>
> For example : Security logging or auditing and then we can decide wether the method should proceed or not.

Activating Using Priority Annotation

In the context of CDI interceptors, priority is a way to specify the order in which interceptors are executed when multiple interceptors are applied to a single method or class.

Interceptors with higher priority values are executed before those with lower priority values. The default priority value for an interceptor is 1000. You can specify a different priority value by using the `@Priority` annotation, which takes an integer value as its argument.

Priority can be also declared as Integers.

There are some common conventions that you can follow. Here are some common priority values and what they might represent:

- High priority (e.g. 1000 or higher): These interceptors should be executed before most other interceptors. They might handle authentication or authorization logic, for example.
- Medium priority (e.g. 500 to 999): These interceptors might handle general-purpose cross-cutting concerns like logging or exception handling.
- Low priority (e.g. 0 to 499): These interceptors might handle less important cross-cutting concerns like caching or performance optimization.

Let's Run the example!

We used our interceptor (`@Logged`) in :

```
@Stateless // simple Stateless EJB class
public class AuditedService {
    @Inject
    private Logger logger;

    // This method will only be called after the Logged Interceptor has returned
i.e. InvocationContext=>proceed
    // This annotation could also be put on the class, making every method of the
class intercepted
    @Logged
    public void auditedMethod() {
    logger.log (Level.INFO, "OK so we are able to call this method after auditing
took place") ;
    }
}
```

then, we used `AuditedService` bean in :

```
@Web
public class ScopesBean implements {
    @lnject
    private AuditedService auditedService;

    // Producer object
    @lnject
    private Logger logger;

    // Lifecyle callback
    @PostConstruct
    private void init() {
    auditedService.auditedMethod(); // we called the intercepted method here <----
    logger.log(Level.INFO, "**********************Scopes bean
called**********************");
}
```

Done

☑ Day 9/90 ==> JavaEE: Day 9/90 - Context and Dependency Injection (CDI)

Day 10/90

Date : 16-April-2023

no progress

Done

☑ Day 10/90 ==> JavaEE: Day 10&11/90 - Context and Dependency Injection (CDI)

Day 11/90

Date : 17-April-2023

no progress

Done

- ☑ Day 11/90 ==> [JavaEE: Day 10&11/90 - Context and Dependency Injection (CDI)](#)

Day 12/90

Date : 18-April-2023

CDI - Context & Dependency Injection

CDI Events

In the CDI API, an event is a mechanism for loosely coupling components in an application. It allows one component to notify other components that something of interest has occurred, without those components needing to know anything about each other.

An event in CDI is represented by an object that carries information about the occurrence that triggered the event. When an event is fired, any observer methods that have been registered to listen for that event are called with the event object as a parameter.

Observer methods can be defined in any bean that is managed by the CDI container, and they can be annotated with the `@Observes` annotation to indicate which events they should listen for. In this way, CDI provides a flexible and extensible way to handle decoupled communication between components in an application.

> It is a way to be able to communicate a proportion of application without compile time dependency. An event will be fired with a passed payload, and then we'll have an observer expecting the payload.

Event Interface

The payload that we're going to pass to the `Event interface`. Simple POJO class.

```java
public class Payload {
    private String email;
    private LocalDateTime loginTime;
    // getters & setters & constructors ...
}
```

```java
@Web //we created this qualifier (annotation/interface annotation) -->
@RequestScoped, @Named (for JSF)
public class EventBean {
    @Inject
    private User user; //another POJO with email & pass

    @Inject
    Event<Payload> plainEvent;

    @Inject
    @PopularStand //custom qualifire with --> @Qualifier
    private Event<Payload> eventDataEvent;

    @Inject
    @Admin
    private Event<Payload> conditionalEvent;

    public void login() {
        //Do credentials check and logic, then fire the event
        //someSecurityManager.loginUser(user.getEmail(), user.getPassword());

        plainEvent.fire(new Payload(user.getEmail()), user.getPassword());

        LocatDateTime now = LocalDateTime.now();
        System.out.println(now);

        eventDataEvent.fire(new LocalDateTime.now()));
        fireAsync = eventDataEvent.fireAsync(new EventData(user.getEmait(), long
secs = ChronoUnit.SECONDS.between(now, LocalDateTime.now());
        System.out.println("It took us this number of seconds to login" + secs);

        //Qualified Observer
        conditionatEvent.fire(new LocatDateTime.now())) ;
        LocalDateTime.now()));
    }
}
```

the observer :

```java
@RequestScoped
public class EventObserver implements Serializable {
    @Inject
    private Logger logger;

    void plainEvent(@Observes Payload payload) {
        //persist in db, sent to another application outside your app
        //essentially you can do whatever you want with the event data here
        //we will just log it
```

```java
        logger.log(Level.INFO, "User {0} logged-in at {1}. Logged from
PLAIN_EVENT_OBSERVER", new Object[] {payload.getEmail(), payload.getLoginTime()});
        try {
            Thread.sleep(6000);
        } catch (InterruptedException e) {
            logger.log(Level.SEVERE, null, e);
        }
    }

    void userLoggedIn(@Observes @PopularStand Payload payload) {
        //persist in db, sent to another application outside your app
        //essentially you can do whatever you want with the event data here
        //we will just log it
        logger.log(Level.INFO, "User {0} logged in at {1}", new Object[]
{payload.getEmail(), payload.getLoginTime()});
        try {
            Thread.sleep(6000);
        } catch (InterruptedException e) {
            logger.log(Level.SEVERE, null, e);
        }
    }

    void asyncObserver(@ObservesAsync @PopularStand Payload payload) {
        //persist in db, sent to another application outside your app
        //essentially you can do whatever you want with the event data here
        //we will just log it
        logger.log(Level.INFO, "User {0} logged in at {1}", new Object[]
{payload.getEmail(), payload.getLoginTime()});
        try {
            Thread.sleep(6000);
        } catch (InterruptedException e) {
            logger.log(Level.SEVERE, null, e);
        }
    }

    void conditionalObserver(@Observes(notifyObserver = Reception.IF_EXISTS, during
= TransactionPhase.AFTER_COMPLETEION) @Admin Payload payload) {
        logger.log(Level.INFO, "The CEO {0} logged in at {1}", new Object[]
{payload.getEmail(), payload.getLoginTime()});
    }
}
```

> - Observers must be of type `void`.

Plain Event

From previous example, `plainEvent field in bean` :

When an event is fired, **any** observer that observing for that particular event, will get notified.

From previous example, `plainEvent observer method`

the observers will observe the event, any event of that particular payload type. Anywhere. Simple as that.

This is not recommended when we have an event, and multiple observers, but you don't want all of them invoked when you want to fire this event. You may want to invoke only one of them. In this case, we use the concept of qualifiers.

Qualifying Events

Qualified events are events that are just labeled with qualifiers, where you tell CDI runtime how one or more things are related.

from previous example `in observer class --> userLoggedIn` :

when we fire `eventDataEvent`, the `userLoggedIn` method will get invoked, but the `plainEvent` method won't get invoked because it does not have the particular qualifier, in our case `@PopularStand`.

Conditional Observers

Out of the box, observers will be invoked once an event is fired. Once the event is fired, the container will look for that particular event's observer. There are situations where you want to invoke an observer conditionally. We can do that using notifier construct (previous example : `observer class -> conditionalObserver(...)`)

```
void conditionalObserver(@Observes(notifyObserver = Reception.IF_EXISTS, //we're
telling the container to only invoke this particular qualifier if there is a
contextual bean that declaring the event observer in the context

during = TransactionPhase.AFTER_COMPLETEION) //we're also telling to also invoke it
if there is a transaction and the transaction succeeds
                                ))
```

Async Events

we use `event.fireAsync(the payload)`.

this object returns CompleteionStage of type Payload `(CompletionStage<Payload>)` object.

and for the observer `void asyncObserver(@ObservesAsync)`.

Prioritizing Observer Method Invocation

use `@Priority(...)`

we can replace `...` with :

1. simple integers (the higher the number, the lower the priority)
2. or we can use CDI Interceptor API (recommended)

   `@Priority(Interceptor.Priority.APPLICATION + 200)`

ʕ⊃ ◕_◕ʔ⊃Summary of CDI API

What I have learned from CDI API

- **Bean discovery mode :** there is a process where the container scans your applciations archive to discover beans to be managed.

- **bean.xml file :** which is what we use to set bean discovery mode. Out of the box, bean discovery mode is set to `annotated` which means CDI container will only manage beans that are annotated with specific CDI annotations.

- **Container :** It is the runtime environment where it manages instantiaion of beans.

- **Beans & Contextual instances :** beans are just the templates, java classes from which contextual instances are created.

- **Injection point :** The class, that the beans are being injected to. Field, constructor, method injection.

- **Lifecycle callbacks :** Mehtods that get invoked at the lifecycle of bean. PostConstruct, PreDestroy.

- **Qualifiers :** We use qualifiers to link certain things together so that the container can manage based on our need. For avoiding ambiguous beans.

- **Stereotypes :** Stereotypes are interfaces grouping together commonly used CDI annotations into one, so that when you use that one particular stereotype, you are using all those other API annnotations that we put together.

- **Context & Scopes :** A <u>Scope</u> defines the context in which a bean instance exists, and thus determines how long an instance will be preserved and when it will be destroyed. <u>Contexts</u> in CDI refer to the runtime environment that manages the lifecycle of a bean instance. `@ApplicationScoped, @RequestScoped, @SessionScoped, @Dependent`.

- **Producer :**

- **Interceptors :** It is a way of implementing cross cutting concerns.

- **Event :** Helps us to write Reactive application.

Done

☑ Day 12/90 ==> [JavaEE: Day 12/90 - Context and Dependency Injection (CDI)](#)

Day 13/90

Date : 19-April-2023

Java Persistence API (JPA)

JPA stands for Java Persistence API, and it is a framework in JavaEE (also in Spring framework) that makes it easier for developers to store and retrieve data from databases. Think of it as a way to interact with a database without having to write SQL code directly. This can save time and make it easier to manage your data persistence layer.

What we are going to learn :

- Object Relational Mapping
- Entity Manger
- Query Language
- Advance JPA

Setting up Payara Server

Download the payara server 5, full version and add it to the IDE. It'll show the name as Glassfish in the IDE.

JPA Entity

Any simple Java POJO class can be JPA entity. We just need to mark it with `@Entity` and add a field for a unique identifier i.e. `@Id`.

Every instance of jpa entity class will reperesent as a row in db, and id will act as a unique identifier for that instance like student id for every student in a school.

```java
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Tax {
    @Id
    private Long id;
    //other fields & setters, getters, constructors
}
```

Customizing Table Mapping

`@Table(name = "any_name")` : customize the name of the class that is going to be saved as table.

`@Table(name = "any_name", schema = "HR")` : It will act as a prefix for the table like => `HR.any_name`

Using Super Classes

In a Enterprise application, we'll have multiple classes that map to the table. All these classes have a unique identifiers that are marked with `@Id` . Since one of the tenets of software developement is dry (don't repeat yourself), we can group all these shared and common property id's into one and extend that class all across the entities.

- create an abstract class

- mark it with `@MappedSupperClass` : this makes the class abstract i.e. there won't be any table for this class in db. Put all the properties in the marked class, into the classes that extend it.

- provide the id property

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@MappedSuperclass
public abstract class AbstractEntityID {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    protected String userEmail;

    @Version
    protected Long version;
}
```

Overriding Super Class Field

We want to override the id in the `AbstractEntityID` and give it another name. To do that :

```
@OverrideAttribute(name = "name_of_the_field", column = @Column(name = "taxId"))
@Entity
public class Tax extends AbstractEntityID {
    ...
}
```

Mapping Simple Java Types

JavaEE JPA API provides an easy and efficient way to map simple Java types to corresponding database columns. Simple Java types such as boolean, integer, string etc. can be easily mapped to their respective column types in the database using annotations like `@Column` and `@Basic` .

For example, if you have a Java class with a String field named "name", you can annotate it with `@Column(name="NAME")` to map it to a database column named "NAME". Similarly, you can use `@Basic` annotation to specify default mapping properties for basic types. `@Basic` annotation is optional.

If you don't provide the `@Column(name)` , JPA will give the default name i.e. the name of the java field.

Transient Fields

In Java EE JPA API, a transient field is a field that is marked with the "transient" keyword and is not persisted to the database. This means that when an entity object is saved or retrieved from the database, any fields marked as transient are ignored. Transient fields are often used for storing data that is not relevant to the persistence of the object, such as derived values or temporary variables. However, it is important to note that transient fields can still be serialized and deserialized along with the entity object, so care should be taken when using them in distributed systems.

Field Access Type

In Java EE JPA API, the `Field Access Type` is a way to define how entity class fields should be accessed by the JPA provider. There are two types of access: Field access and Property access.

With `Field Access`, the JPA provider accesses the fields directly, using reflection. In this case, all entity class fields must be declared as `private` and must not have any custom get/set methods. Here is an example:

```java
Copy Code@Entity
@Access(AccessType.FIELD)
public class Employee {
    @Id
    private Long id;

    private String firstName;
    private String lastName;

    // Constructors, getters and setters
}
```

With `Property Access`, the JPA provider accesses the properties of the class using their corresponding getter and setter methods. In this case, all entity class fields must be declared as `private`, and the corresponding getter and setter methods must be implemented. Here is an example:

```java
Copy Code@Entity
@Access(AccessType.PROPERTY)
public class Employee {
    private Long id;

    private String firstName;
    private String lastName;

    @Id
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
```

```
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    // Other methods
}
```

It is important to note that the default access type is `Field Access`. However, you can explicitly specify the access type using the `@Access` annotation.

Mapping Enumerator Type

In Java EE JPA API, enum types can be mapped to database columns using the `@Enumerated` annotation. This annotation can be applied to a field or getter method of an entity class to specify how the enum value should be persisted in the database.

The `@Enumerated` annotation supports two modes of conversion: `ORDINAL` and `STRING`. When the `ORDINAL` mode is used, the enum value is stored as its ordinal position in the enum declaration (starting from 0). In the `STRING` mode, the enum value is stored as its string representation (i.e., the name of the enum constant).

It is important to note that the `ORDINAL` mode should only be used if the order of the enum constants is unlikely to change, as changing their order would affect the data stored in the database. (like adding another constant in the enum class.)

```
@Entity
public class Tax {
    @Id
    @GeneratedValue(strategy = IDENTITY)
    private Long id;

    @Enumerated(EnumeratedType.STRING)
    private CustomEnumClass customEnumClass;

    ...
}
```

Day 14/90

Java Persistence API (JPA) (continue)

Mapping Large Objects (e.g. images)

The Java EE JPA API provides a powerful and flexible way to map large objects in a relational database. This is done using the `@Lob` annotation, which can be used on fields of type `String`, `byte[]`, or `Serializable`.

When a field is annotated with `@Lob`, JPA will automatically create a separate table for the large object, and store a reference to that object in the original table. This allows for efficient storage and retrieval of large objects, while still maintaining the benefits of a relational database.

It's worth noting that while `@Lob` is convenient, it should be used sparingly as it can have performance implications. For very large objects or frequent updates, it may be better to store the data in a separate table altogether.

```java
// ...
@Entity
public class Demo {
    @Lob
    private byte[] picture;
    // ...
}
```

It is better to make the large objects as lazy fetching.

```java
...
@Lob
@Basic(fetch = FetchType.LAZY)
private byte[] picture;
...
```

Lazy & Eager Fetching Of Entity State

Fetching is the process of retrieving data from the database and loading it into memory so it can be used by your application. In JPA, there are two types of fetching strategies: Lazy and Eager.

Lazy fetching means that JPA will only load the data when it is actually needed. For example, if you have an object that has a list of related objects, JPA will not load that list until you try to access it. This can help keep your application running fast because it saves on unnecessary loading of data.

Eager fetching, on the other hand, loads all of the data for an object and its related objects when the object is loaded from the database. This can be beneficial if you know that you will need all of the data anyway, but it can also slow down your application if you are working with large amounts of data.

In general, it's a good idea to use lazy fetching whenever you can. Lazy loading can help make your application more responsive and efficient because it only loads data when it's needed. However, there may be cases where eager loading makes more sense. For example, if you know that you will always need certain data when you load an object, eager fetching could be more efficient because you won't need to load the data later.

To implement lazy or eager fetching in JPA, you can use annotations in your code. For example, to specify that a relationship between two entities should be lazily fetched, you would use the `@OneToMany` or `@ManyToOne` annotation with the fetch attribute set to `FetchType.LAZY`. Conversely, to specify that a relationship should be fetched eagerly, you would set the fetch attribute to FetchType.EAGER.

Mapping Java 8 DateTime Types

In Java EE, the Java Persistence API (JPA) provides a standard way to map Java objects to relational databases. When mapping date and time values in Java 8, JPA can use the new `java.time` package which offers several new classes for representing date and time values.

Mapping Embeddable classes

In Java EE JPA API, you can use something called an "embeddable class" to represent complex data types within your entity classes. These embeddable classes are non-entity classes that you can embed within your entity classes.

To map an embeddable class, you just need to add the `@Embeddable` annotation at the beginning of the class definition. Then, within your entity class, you can use the `@Embedded` annotation on the corresponding field to map the embeddable class.

You can also use other JPA annotations such as `@Column` or `@Temporal` within your embeddable class to specify how its fields should be mapped.

Using embeddable classes can help you manage complex data structures within your entities without having to create additional database tables.

> Embeddable is essentially an object that has no identity on its own, and it becomes a part of the class in which it is embedded.

Done

- ☑ Day 14/90 ===> JavaEE: Day 14/90 - Java Persistence API (JPA)