

Contents

Contents

Day 1/90

Understanding JavaEE - The theory of JavaEE

What is JavaEE?

-What is an Application Server?

Examples of JavaEE Application Server

What is a JSR

JSR Examples

What is a Reference Implementation?

What is JakartaEE?

JavaEE/JakartaEE vs Spring Framework

Done

Day 2/90

Setup

Deploy

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

Done

Day 3/90

Continue of previous lesson (Overview of JavaEE)

Persistence Unit

`@Transactional`

RESTful Endpoint

Deployment

Done

Day 4/90

Zero progress

Done

Day 5/90

Continue of previous lesson (Overview of JavaEE)

Packaging & Deploying web application on payara micro - method 1

Validation

JavaEE Uber Jar

Packaging & Deploying with Payara micro - method 2

 Summarize

Done

Day 6/90

CDI - Context & Dependency Injection

What is Dependency Injection

Inversion of Control

 Summarize

CDI Features

Type safe Dependency

e.g.

Lifecycle Context

e.g.

Interceptors

e.g.

Events

e.g.

Service Provider Interface (SPI)

e.g.

Bean Discovery Mode

CDI Container

Done

Day 7/90

CDI - Context & Dependency Injection

What are Beans and Contextual Instances?

CDI Injection Point

Field Injection

Constructor Injection

Method Injection

CDI Lifecycle Callback

PostConstruct

PreDestroy

Managed Beans & Bean Types

CDI Qualifiers

Creating Qualifiers with Values

CDI Stereotypes

@Named

CDI Scopes & Contexts

@Dependent

@RequestScope

@SessionScoped

@ApplicationScoped

@ConversationScoped

Done

Day 8/90

CDI - Context & Dependency Injection

Context & Scopes in action

↪ ●_● ↪ Summarize

CDI Producers

Scoping Returned Beans

Field Producers

Qualifying Beans

Disposers

↪ ●_● ↪ Summarize

Done

Day 9/90

CDI - Context & Dependency Injection

CDI Interceptors

Activating Using Priority Annotation

Let's Run the example!

Done

Day 10/90

Done

Day 11/90

Done

Day 12/90

CDI - Context & Dependency Injection

CDI Events

Event Interface

Plain Event

Qualifying Events

Conditional Observers

Async Events

Prioritizing Observer Method Invocation

 Summary of CDI API

Done

Day 13/90

Java Persistence API (JPA)

Setting up Payara Server

JPA Entity

Customizing Table Mapping

Using Super Classes

Overriding Super Class Field

Mapping Simple Java Types

Transient Fields

Field Access Type

Mapping Enumerator Type

Done

Day 14/90

Java Persistence API (JPA) (continue)

Mapping Large Objects (e.g. images)

Lazy & Eager Fetching Of Entity State

Mapping Java 8 DateTime Types

Mapping Embeddable classes

Done

Day 15/90

Java Persistence API (JPA) (continue)

Mapping Primary Keys

Auto Primary Key Generation Strategy

Entity Relationship Mapping

Roles

Directionality

Cardinality

Ordinality

Entity Relationship Mappings

Ownership of Relationships

Unidirectional

Bidirectional

@ManyToOne

@OneToOne

Done

Day 16/90

Java Persistence API (JPA) (continue)

Collection Valued Relationship

@OneToMany

@ManyToMany

Fetch Mode

Collection Mapping Of Embeddable Objects and Collection Table

Ordering The Contents Of a Persistable Collection

Mapping Persistable Maps

Done

Day 25/90

Day 26/90

Keying Persistable Maps by *Entities*

Architecture of EJB

3 Message-Driven Beans :

Done

Elements Of Persistence Unit

Combined Path Expressions

Done

Where Clause - Like Operator

Done

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 `jakarta.*` 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

```
<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
```

- note : 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.


```

...
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!

```
<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`

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 their dependencies supplied to them instead of creating 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 objects and dependencies in our application.

e.g. instead of saying `Foo foo = new Foo();` 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:

```
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.



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:


```
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:

```
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:

```

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:

```

@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:

```

@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 is 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 the 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:

```
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:

```
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 go in, and come 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.

```
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.

```
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.

```

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:

```
@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:

```
@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:

```
@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.

```
public interface 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.

```
@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` :

```
// @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);
    }
}
```

```
// @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 :

```

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 soldiersSalute;

    // 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.RUNTIME)
@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 annotations 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 : Application scoped, will create a single contextual instance of a bean type and associate with the lifetime of the application itself.

It is basically a singleton that last throughout 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 generally 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 7/90 ==> [JavaEE: Day 7/90 - Context and Dependency Injection \(CDI\)](#)

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.

```
@RequestScoped
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 ...
}
```

```
}
```

🔗🔗🔗🔗 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:

```
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
@PersistenceContext
EntityManager entityManager;

// now wherever 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.



CDI Producer

- is a way to transform mostly classes that we don't own into CDI managed beans.
- We can use either producer method or field, 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 your 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 behavior of 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 whether 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 {
    @Inject
    private AuditedService auditedService;

    // Producer object
    @Inject
    private Logger logger;

    // Lifecycle 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.

```

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

```

```

@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.getEmail(), 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 :

```

@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
    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_COMPLETION) //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 `CompletionStage` 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)`

Summary of CDI API

What I have learned from CDI API

- **Bean discovery mode** : there is a process where the container scans your applications 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 instantiation 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** : Methods 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 annotations that we put together.
- **Context & Scopes** : 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.
`@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 represent 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.

```
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 development 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 `@MappedSuperClass` : 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:

```
javaCopy 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:

```
javaCopy 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(EnumType.STRING)
    private CustomEnumClass customEnumClass;

    ...
}

```

Done

✓ Day 13/90 ==> [JavaEE: Day 13/90 - Java Persistence API \(JPA\)](#)

Day 14/90

Date : 20-April-2023

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.

```

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

```

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

```
...
@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 \(JPA\)](#)

Day 15/90

Date : 24-April-2023

Java Persistence API (JPA) (continue)

Mapping Primary Keys

So, in Java EE's JPA API, we have something called primary keys `ID`. Primary keys are like your unique ID on a social media platform - it uniquely identifies you from everyone else.

JPA makes it easy to work with primary keys by letting us use the `@Id` annotation. It's like putting a stamp on a letter to make sure it goes to the right address 📧. We can also use the `@GeneratedValue` annotation to automatically generate primary key values, like how your social media platform assigns you a unique username when you sign up.

If we need to use multiple fields/columns to create our primary key, we can use the `@EmbeddedId` or `@IdClass` annotations 🧡.

Overall, JPA's support for primary keys makes sure that each entity in our database has a unique identity and helps us keep track of them easily 👍.

What types can we use for our id !?

Good question! 🤔

In JPA, we can use different types for our primary keys `ID`. Here are a few examples:

- ◆ **Numeric types:** We can use numeric types like `int` or `Long` as our primary keys. This is like using a number to identify something - kind of like a ticket number 🎫.
- ◆ **String type:** We can also use the "String" type for our primary keys. This is like using a word or phrase to identify something - like a username on a social media platform 🔍.
- ◆ **UUID type:** Another option is to use the `UUID` (Universally Unique Identifier) type for our primary keys. This generates a unique identifier that is unlikely to be duplicated, kind of like a fingerprint 👍.
- ◆ **Custom type:** Finally, we can even create our own custom data type for our primary keys if none of the built-in types suit our needs. This is like creating your own secret code to identify something 🤖.

Each type has its own advantages and disadvantages, so it's important to choose the one that best fits our needs based on factors like performance, uniqueness, and readability.

Auto Primary Key Generation Strategy

In the Java EE's JPA API, we can use something called an "Auto Primary Key Generation Strategy" to automatically generate primary key values for our entities. This can save us time and effort by eliminating the need to manually assign primary key values 🇺🇸.

There are several different strategies that we can choose from:

◆ **IDENTITY:** This strategy relies on an identity column in the database to automatically generate primary key values. It's like getting a ticket number at a deli 🇺🇸. Here's an example:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
```

This strategy is typically used when you're working with databases that support identity columns, such as SQL Server or MySQL. It's a simple strategy that doesn't require any additional configuration beyond annotating your ID field with `@GeneratedValue(strategy = GenerationType.IDENTITY)`.

◆ **SEQUENCE:** This strategy uses a database sequence to generate primary key values. It's like getting a numbered ticket at a carnival ride 🎡. Here's an example:

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
    generator = "product_seq")
@SequenceGenerator(name = "product_seq",
    sequenceName = "product_sequence",
    allocationSize = 1)
private Long id;
```

This strategy is often used when you're working with databases that support sequences, such as Oracle or PostgreSQL. It allows for more control over how primary keys are generated and can be useful if you need to generate IDs across multiple tables using the same sequence.

◆ **TABLE:** This strategy uses a separate database table to keep track of primary key values. It's like keeping a tally of how many people have gone through a turnstile 🚶. Here's an example:

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE,
    generator = "product_gen")
@TableGenerator(name = "product_gen",
    table = "id_generator",
    pkColumnName = "gen_name",
    valueColumnName = "gen_val",
    allocationSize = 1)
private Long id;
```


This strategy is useful when you're working with databases that don't support either identity columns or sequences, or if you need to generate primary keys across multiple nodes in a distributed system. It creates a separate table to keep track of primary key values, which can make it slower than other strategies but also more reliable in certain scenarios.

♦ **AUTO:** This strategy allows the JPA provider to automatically choose the most appropriate strategy based on the database being used. It's like asking a robot to pick out the best strategy for generating primary key values 🤖.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

This strategy is typically used when you're not sure what type of database you'll be working with or if you want the JPA provider to automatically choose the best strategy based on the database being used. It's a good choice if you want to write code that's more portable across different database systems.

👉 Each strategy has its own strengths and weaknesses, so it's important to choose the one that best fits our needs based on factors like performance, scalability, and database compatibility.

By using auto primary key generation strategies, we can focus on building our application logic without worrying about the complexities of managing primary keys 👍.

🛠️ During the development phase, we can use a "persistence provider" tool to generate a database schema for our application. Think of this as creating a blueprint or plan for the structure of our database.

📦 In a JavaEE JPA API application, the persistence provider is responsible for managing the communication between our application and the database. The JavaEE JPA specification defines a standard set of interfaces that a persistence provider must implement. Some examples of popular persistence providers in the JavaEE ecosystem include Hibernate, EclipseLink, and Apache OpenJPA.

🇮🇹 However, once our application is ready for production, we want the database itself to handle the schema generation for us. This means that we don't need to rely on the persistence provider tool anymore.

💻 Instead, we can configure our JavaEE JPA API application to work with the production database directly. This allows the database to automatically create and update its own schema based on the data our application sends to it.

🇬🇧 This is a more efficient and reliable way to manage our database's schema because it removes the need for an extra tool during production.

Entity Relationship Mapping

ERM (Entity Relationship Mapping) is a technique used to model data in a database. It helps us represent entities (like people, products, or orders) as tables in a relational database. We also use ERM to show the relationships between entities.

Think of ERM like a family tree 🌳! Just like how family members are related to each other, entities in ERM are related to each other. For example, a person can be related to another person as their parent, sibling or spouse. In ERM, we use different types of relationships like `One-to-One`, `One-to-Many`, and `Many-to-Many` to represent these connections between entities.

For example, imagine we have an online store application, where customers can place orders for products. With JPA, we can create Java classes for our Customer and Order entities, and use annotations to define the relationship between them. This way, we can easily retrieve all the orders placed by a particular customer or find out which products were ordered together in the same order.

Roles

In ERM, we use different types of relationships like One-to-One, One-to-Many, and Many-to-Many to represent the connections between entities. But in addition to relationships, entities can also have roles within those relationships.

Think of roles like characters in a play 🎭! Just like how each character has a specific role to play in the story, entities in ERM can have specific roles within a relationship. For example, if we have a "Person" entity and a "Role" entity, the "Person" could have a "Manager" role or an "Employee" role within the "Role" entity.

Directionality

In ERM, we use different types of relationships like One-to-One, One-to-Many, and Many-to-Many to represent the connections between entities. But these relationships can also have directionality - meaning that they can be one-way or two-way.

Think of directionality like a one-way street 🚗! Just like how cars can only travel in one direction along a one-way street, some relationships in ERM can only go one way. For example, if we have a "Person" entity and an "Address" entity, the relationship between them might be one-way - meaning that we can find the address for a person, but not vice versa.

For example, imagine we have a social media application where users can follow other users, but being followed doesn't necessarily imply a reciprocal relationship. With JPA, we can create Java classes for our User and Follower entities, and use annotations to define a one-way relationship between them.

Another example, imagine a person and his house 🏠. The person knows the address of his house, but does it make sense for the house to have a relationship with its resident 🤖!?

Cardinality

In ERM, we use different types of relationships like One-to-One, One-to-Many, and Many-to-Many to represent the connections between entities. But these relationships can also have cardinality - meaning that they can be mandatory or optional.

Think of cardinality like attending an event 🎉! Just like how some events require you to attend, while others are optional, some relationships in ERM can be mandatory, while others can be optional. For example, if we have a "Person" entity and a "Passport" entity, the relationship between them might be mandatory - meaning that every person must have a passport.

For example, imagine we have a school management application where students can enroll in courses, but not every course is mandatory for every student.

It means, how many entities are on which end of relationships.

Ordinality

Think of it this way: in real life, you have relationships with people. 👤 You might have a best friend, a sibling, a parent, or a significant other. Each of these relationships has a certain "strength" - for example, your relationship with your best friend might be closer than your relationship with a coworker.

In the same way, in Java EE JPA API, entities are related to each other. 💖 And just like in real life, these relationships have different strengths - or "ordinalities."

For example, let's say you have two entities: "Customer" and "Order". 📦 A customer can have many orders, but an order can only belong to one customer. This is an example of a "one-to-many" relationship - the "one" side being the customer, and the "many" side being the orders.

It means, other end of the relationship, should be available or not.

Entity Relationship Mappings

There are four types of relationships in ERM:

1 Many-to-One (N:1): This relationship is when many instances of one entity are related to a single instance of another entity. For example, think about students and classes. Each student can enroll in multiple classes, but each class can have many students enrolled in it. In this scenario, we would say that there is a Many-to-One relationship between Student and Class entities, because many students can be associated with a single class.

2 One-to-One (1:1) : This relationship is when one entity is connected to only one other entity. For example, a person can have only one passport, and a passport belongs to only one person.

3 One-to-Many (1:N) : This relationship is when one entity is related to many other entities. For example, a customer can place many orders, but each order belongs to only one customer.

4 Many-to-Many (N:M) : This relationship is when many entities are related to many other entities. For example, a book can be written by many authors, and each author can write many books.

To implement these relationships using JPA, we use annotations like @OneToOne, @OneToMany, and @ManyToMany. These annotations are added to the fields or methods of our Java classes that represent entities 🤖.

Ownership of Relationships

🏠 Ownership in JPA API can be determined using two methods:

1 Unidirectional Relationships: In a unidirectional relationship, one entity owns the relationship and references the other entity without being referenced back. This is similar to a one-way street where traffic flows in only one direction.

2 Bidirectional Relationships: In a bidirectional relationship, both entities reference each other. This is similar to a two-way street where traffic flows in both directions.

In JPA API, ownership of an entity or relationship is determined based on the presence or absence of the `mappedBy` attribute in the relationship annotations. If an entity includes the `mappedBy` attribute, it means that the relationship is owned by the other entity in the relationship (we can also determine ownership via `@JoinColumn`).

Unidirectional

👁️ Let's see an example of how we could represent a unidirectional One-to-Many relationship between `Order` and `Item` entities:

```
javaCopy Code@Entity
public class Order {
    @Id
    private Long id;

    // One-to-many relationship with Item
    @OneToMany
    private List<Item> items;

    // getters and setters omitted for brevity
}

@Entity
public class Item {
    @Id
    private Long id;
    private String name;

    // No relationship mapping here!

    // getters and setters omitted for brevity
}
```

📦 In this example, each `Order` entity has many `Item` entities associated with it, represented by the `items` field with the `@OneToMany` annotation. Because there is no `mappedBy` attribute in the `@OneToMany` annotation, it means that the relationship is unidirectional and `Order` entity owns the relationship.

🗄️ Now, let's see how these entities might be represented in a database:

Order Table

id	other columns...
1	...
2	...
3	...

Item Table

id	name	order_id
1	Item A	1
2	Item B	1
3	Item C	2
4	Item D	3
5	Item E	3

🔗 In this example, we can see how each item is associated with a single order through the `order_id` foreign key column in the `Item` table. Because `Order` entity owns the relationship, the `Item` table includes the foreign key column that references the primary key of the matching order in the `Order` table.

Bidirectional

👁️ here's an example of how we could represent a bidirectional One-to-Many relationship between Customer and Order entities:

```
@Entity
public class Customer {
    @Id
    private Long id;

    // one-to-many relationship with Order
    @OneToMany(mappedBy = "customer")
    private List<Order> orders;


    // getters and setters omitted for brevity
}


@Entity
public class Order {
    @Id
    private Long id;

    // Many-to-one relationship with Customer
    @ManyToOne
    private Customer customer;

    // other columns...

    // getters and setters omitted for brevity
}
```

 In this example, each Customer entity has many Order entities associated with it, represented by the orders field with the `@OneToMany` annotation. However, unlike the previous example, this time we also have the `mappedBy` attribute set to "customer" in the `@OneToMany` annotation.


 Now, let's see how these entities might be represented in a database:

Customer Table

id	other columns ...
1	...
2	...


Order Table


id	customer_id	other_columns ...
1	1	...
2	1	...
3	2	...

 In this example, we can see how each order is associated with a single customer through the `customer_id` foreign key column in the Order table. The `mappedBy` attribute in the `@OneToMany` annotation specifies that the relationship is bidirectional, and that the owning side of the relationship is the `order` entity.

On the other hand, the `@ManyToOne` annotation in the `Order` entity specifies the non-owning side of the relationship, where each `Order` entity has one `Customer` associated with it. This is represented by the `customer` field, which has a foreign key that references the `id` primary key of the matching customer in the Customer table.

@ManyToOne

Think of it this way: in real life, you might have a boss who manages a team of employees.  The boss is the "one" side, while the employees are the "many" side. Each employee belongs to exactly one boss, but each boss can manage many employees.

In Java EE JPA API, we can represent this relationship using the `@ManyToOne` annotation.  Let's say we have two entities: "Employee" and "Boss". An employee belongs to exactly one boss, so we'll add a `@ManyToOne` annotation to the Boss field in the Employee class.

here's an example!

Let's say we have two entities: "Product" and "Category". Each product belongs to exactly one category, but each category can have many products. We'll use the `@ManyToOne` annotation to represent this relationship.

Here's the code for the Product entity:

```
@Entity
public class Product {
    @Id
    private Long id;

    private String name;

    // Many-to-one relationship with Category
    @ManyToOne
    private Category category;

    // getters and setters omitted for brevity
}
```

And here's the code for the Category entity:

```
@Entity
public class Category {
    @Id
    private Long id;

    private String name;

    // One-to-many relationship with Product
    @OneToMany(mappedBy = "category")
    private List<Product> products;

    // getters and setters omitted for brevity
}
```

In this example, each Product entity has a single Category associated with it, represented by the `category` field with the `@ManyToOne` annotation. Meanwhile, each Category entity has a list of Products associated with it, represented by the `products` field with the `@OneToMany` annotation (which we'll cover in another section!).

here's an example of how the two tables might look based on the code provided:

Product Table

id	name	category_id
1	Product A	1
2	Product B	2
3	Product C	1

Category Table

id	name
1	Category X
2	Category Y

how do we determine the ownership of the entities? In our case, is Category the owner, or the Product? It is determined by looking which db has the foreign key column.

@OneToOne

💛 One of the types of relationship annotations is called `@OneToOne`. `@OneToOne` represents a relationship where one entity is related to exactly one other entity.

For example, think of a romantic couple! Each person has exactly one partner (at least, we hope so!). In database terms, we might have two tables: "Person" and "Partner". Each person has exactly one partner, and each partner belongs to exactly one person.

💕 Let's see an example of how we could represent this relationship in our code:

```
@Entity
public class Person {
    @Id
    private Long id;
    private String name;

    // One-to-one relationship with Partner
    @OneToOne(mappedBy = "person")
    private Partner partner;

    // getters and setters omitted for brevity
}

@Entity
public class Partner {
    @Id
    private Long id;
    private String name;

    // One-to-one relationship with Person
    @OneToOne
    private Person person;

    // getters and setters omitted for brevity
}
```


💖 In this example, each `Person` entity has exactly one `Partner`, represented by the `partner` field with the `@OneToOne` annotation. Meanwhile, each `Partner` entity also has exactly one `Person`, represented by the `person` field with the `@OneToOne` annotation.

👤 Now, let's imagine we have two people in our database: Alice and Bob. They are a couple, so they each have exactly one partner. In the database, we might have a `Person` table and a `Partner` table:

Person Table

id	name
1	Alice
2	Bob

Partner Table

id	name	person_id
1	Bob	1
2	Alice	2

🎉 In this example, Alice and Bob are each other's partners, and we can see how the `person_id` column in the `Partner` table references the primary key (`id`) of the matching person in the `Person` table.

uhhh... I (the actual author "Seyed Ali") was not the one who made this example and analogy 😊🙏

49

Done

✅ Day 15/90 ==> [JavaEE: Day 15/90 - Java Persistence API \(JPA\)](#)

Day 16/90

Date : 25-April-2023

Java Persistence API (JPA) (continue)

Collection Valued Relationship

A Collection Valued Relationship is when an entity has a relationship with multiple instances of another entity. To illustrate this concept, let's use a real-world example: think of a library 📖. The "Book" entity might have a Collection Valued Relationship with the "Author" entity, because each book can have one or more authors.

There are two ways to implement collection valued relationship : `@OneToMany` & `@ManyToMany`.

@OneToMany

A common type of relationship is `@OneToMany`, which represents a one-to-many relationship between two entities. This means that one entity can be associated with multiple instances of another entity.

📖 A real-world example of this would be a library system, where each book can have many authors. In this case, the Book entity would have a `@OneToMany` relationship with the Author entity.

To define this relationship in code, we use annotations like this:

```
@OneToMany
private Collection<Author> authors;
```

👨👩 Let's say we have a family tree application, where each family member can have multiple pets. In this case, we would define a `@OneToMany` relationship between the FamilyMember entity and the Pet entity.

Here's what the code for our entities might look like:

```
@Entity
public class FamilyMember {
    @Id
    private Long id;

    @OneToMany(mappedBy = "owner")
    private List<Pet> pets;
    // ...
}

@Entity
public class Pet {
    @Id
    private Long id;

    @ManyToOne
    @JoinColumn(name = "owner_id")
    private FamilyMember owner;
    // ...
}
```

In this example, the FamilyMember entity has a `@OneToMany` relationship with the Pet entity. The "target entity" for this relationship is the Pet class.

The `@OneToMany` annotation is defined on the FamilyMember's pets field. This means that each FamilyMember object can have multiple Pet objects associated with it.

The "ownership of entity in the database" is represented by the `mappedBy` attribute in our example. This tells JPA that the owning side of the relationship is the Pet entity, and that the "owner" field in the Pet class should be used as the foreign key in the database table for the relationship.

So, in our example, the Pet entity "owns" the relationship in the database. This means that the Pet table will have a foreign key column called "owner_id" that references the primary key column of the FamilyMember table.

If we put `@JoinColumn(name = "specify_a_name")`, it will act just like the `mappedBy` attribute.

@ManyToOne

`@ManyToOne` is an annotation used in JPA to define a many-to-many relationship between two entities. This means that an instance of one entity can be associated with multiple instances of another entity, and vice versa.

☀️ Real world analogy: ☀️ Think about a music streaming app like Spotify where a user can have multiple playlists, and each playlist can contain multiple songs. Similarly, a song can be a part of multiple playlists as well. This is a classic example of a many-to-many relationship.

Here's an example of how we can use `@ManyToOne` in JPA:

Let's say we have two entities: "Student" and "Subject". A student can have multiple subjects, and a subject can be taken by multiple students.

To create this relationship, we'll use the `@ManyToOne` annotation in both entities. We'll also need to specify the target entity and the ownership of the entity in the database using `@JoinTable`, `@JoinColumn`, `joinColumns`, and `inverseJoinColumns`.

Here's what it would look like:

```
@Entity
public class Student {
    @Id
    private Long id;
    private String name;

    @ManyToOne
    @JoinTable(
        name = "student_subject",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "subject_id"))
    private List<Subject> subjects; // right now, I'm the owner!

    // getters and setters
}
```

```

@Entity
public class Subject {
    @Id
    private Long id;
    private String name;

    @ManyToMany(mappedBy = "subjects")
    private List<Student> students; // I'm owned by Student 😊

    // getters and setters
}

```

In this example, we're using the `@JoinTable` annotation to specify the name of the relationship table (`student_subject`) and the columns that link the two entities together.

The `joinColumns` attribute specifies the foreign key column in the relationship table that references the primary key of the owning entity (in this case, `student_id`). The "inverseJoinColumns" attribute specifies the foreign key column that references the primary key of the target entity (in this case, `subject_id`).

We also use the `mappedBy` attribute in the `Subject` entity to specify that the relationship is mapped by the `subjects` field in the `Student` entity.

That's it! Now each student can have multiple subjects, and each subject can be taken by multiple students.

Fetch Mode

In JPA, "fetch mode" refers to how related entities are loaded from the database when querying data. There are two types of fetch modes: eager and lazy.

🐱 A good example to understand fetch modes would be fetching data about cats and their owners from a database. Let's say we have one table for cats and another table for owners. Each cat has an owner, and each owner can have multiple cats.

👁️ In eager fetch mode, when we query for a cat, the associated owner information will also be retrieved from the database at the same time. This means that all the data about the cat and its owner will be available in memory immediately, which is convenient if we need to access both sets of data frequently.

👤 However, if we have many cats and their owners in the database, eagerly fetching all the owner data for each cat can result in a lot of unnecessary data being loaded into memory. This can slow down our application and use up resources.

😴 In contrast, lazy fetch mode means that only the data for the requested entity is fetched from the database initially. The associated data is only loaded when it is specifically requested later on. For example, if we initially retrieve data about a cat, the owner information will only be loaded into memory when we try to access it.

🏃 Lazy fetch mode is useful in situations where we have very large amounts of data or where we want to optimize performance by minimizing the amount of data loaded into memory at any given time.

Collection Mapping Of Embeddable Objects and Collection Table

In JPA, it's possible to map collections of embeddable objects to multiple columns in a separate table using the `@ElementCollection` annotation. Each column corresponds to a property in the embeddable object.

🌱 For example, let's say you have a gardening application where users can add their favorite plants to a collection. You might want to capture additional details about each plant, such as the plant's color, size, and type of soil it needs to grow.

🌿 To accomplish this in JPA, you could create an `@Embeddable` class called `PlantDetails` with properties for color, size, and soil type. You could then define a collection of `PlantDetails` in your user entity using the `@ElementCollection` annotation:

```
@Entity
public class User {
    @Id
    private Long id;

    @ElementCollection
    private List<PlantDetails> favoritePlants;
}

@Embeddable
public class PlantDetails {
    private String color;
    private String size;
    private String soilType;
}
```

💡 When your application adds a new plant to a user's collection, JPA will automatically create a new row in the collection table with the corresponding values for each column mapped to the `PlantDetails` properties. 🌻

🌱 Let's continue with our gardening application example. In addition to capturing plant details for each user's favorite plants, let's say we also want to track the number of times each user has watered each plant.

🌿 To accomplish this, we can define a new entity called `Plantwatering`, which includes a reference to the user, a reference to the plant (using the `PlantDetails` embeddable), and a count for the number of times the plant has been watered:

```
javaCopy Code@Entity
public class Plantwatering {
    @Id
    private Long id;

    @ManyToOne
    @JoinColumn(name = "user_id")
```

```

private User user;

@Embedded
@AttributeOverrides({
    @AttributeOverride(name = "color", column = @Column(name = "plant_color")),
    @AttributeOverride(name = "size", column = @Column(name = "plant_size")),
    @AttributeOverride(name = "soilType", column = @Column(name =
"plant_soil_type"))
})
private PlantDetails plantDetails;

private int waterCount;
}

```

🌟 Here, we're using the `@Embedded` annotation along with `@AttributeOverrides` to map the `PlantDetails` properties to separate columns in the `Plantwatering` table. We're also using `@ManyToOne` and `@JoinColumn` to establish a relationship between `Plantwatering` and `User`.

💧 Now, to capture multiple waterings for each plant, we can define a collection of `Plantwatering` entities in our `User` entity using the `@ElementCollection` annotation with `@CollectionTable`:

```

@Entity
public class User {
    @Id
    private Long id;

    @ElementCollection
    @CollectionTable(
        name = "plant_waterings",
        joinColumns = @JoinColumn(name = "user_id")
    )
    private List<Plantwatering> plantwaterings;
}

```

🌱 The `@CollectionTable` annotation here specifies the table name (`plant_waterings`) and the join column that links the collection table to the `user` entity table. With this setup, JPA will automatically create a new row in the `plant_waterings` table each time a user adds a new watering for one of their favorite plants.

Ordering The Contents Of a Persistable Collection

When it comes to ordering the contents of a persistable collection, JPA provides a way to do so using the `@OrderBy` annotation. This annotation can be applied to a collection field in an entity class and specifies the ordering of the elements in the collection.

🌮 Let's imagine we have a `Taco` entity class with a `toppings` field which contains a list of toppings for the taco. We want to display the toppings for each taco in alphabetical order. To achieve this, we can annotate the `toppings` field in the `Taco` entity class with `@OrderBy("name")` where "name" is the name of the property in the `Topping` entity class that we want to sort by.

💻 Here's an example code snippet:

```

@Entity
public class Taco {
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "taco_id")
    @OrderBy("name")
    private List<Topping> toppings;
}


@Entity
public class Topping {
    @Id
    private Long id;
    private String name;
}


```

In this example, the toppings field in the Taco entity class is annotated with `@OrderBy("name")`, which means the toppings will be sorted in ascending order by their name property.

We can also order it by the relationship's attributes.

Mapping Persistable Maps

 In JPA (Java Persistence API), mapping refers to the process of connecting or linking data in a database table to an object-oriented programming language, such as Java.

 Sometimes, we may need to map a map or dictionary-like data structure to a database table. For example, let's say we have a class called Customer, and each customer has a map of their contact details, such as phone numbers and email addresses.

 Here's an example of how we can use JPA annotations to map this data structure:

```

@Entity
public class Customer {
    @Id
    private Long id;

    @ElementCollection
    @MapKeyColumn(name="contact_type")
    @Column(name="contact_detail")
    @CollectionTable(name="customer_contact_details",
joinColumns=@JoinColumn(name="customer_id"))
    private Map<String, String> contactDetails = new HashMap<>();

    // getters and setters
}

```

 Let's break down what's going on here:

- The `@ElementCollection` annotation tells JPA that the `contactDetails` field should be mapped as a collection of embeddable objects.

- The `@MapKeyColumn(name="contact_type")` annotation specifies that the keys in the `contactDetails` map should be stored in a column called "contact_type".
- The `@CollectionTable` annotation specifies that the `contactDetails` collection should be stored in a separate table called "customer_contact_details" with a foreign key column linking it to the `Customer` entity.

🎉 And that's it! Now, JPA knows how to map our `contactDetails` map to a database table.

Using Enums As Persistable Map Keys

💡 Using enums as persistable map keys can be useful when you want to store data in a way that's easy to read and maintain.

🌟 Here's an example of how you can use enums as persistable map keys using the Java EE JPA API:

Let's say you have an enum called "DaysOfWeek" that looks like this:

```
public enum DaysOfWeek {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

You can then create a persistent class called "Schedule" that uses this enum as a key in a map:

```
@Entity
public class Schedule {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @ElementCollection(targetClass = String.class)
    @MapKeyEnumerated(EnumType.STRING)
    @MapKeyColumn(name = "day_of_week")
    @Column(name = "activity")
    private Map<DaysOfWeek, String> activitiesByDayOfWeek;

    // getters and setters
}
```

In this example, we've annotated the `activitiesByDayOfWeek` field with the `@ElementCollection` annotation to indicate that it's a collection of simple types (in this case, strings) rather than another entity.

The `@MapKeyEnumerated` annotation indicates that the keys of the map should be stored as enums and the `@MapKeyColumn` annotation specifies the name of the column that will store these enums.

Now you can create instances of the `Schedule` class and add activities for each day of the week:


```
Schedule weeklySchedule = new Schedule();
weeklySchedule.getActivitiesByDayOfWeek().put(DaysOfWeek.MONDAY, "Go to work");
weeklySchedule.getActivitiesByDayOfWeek().put(DaysOfWeek.TUESDAY, "Attend yoga
class");
// ...and so on
```

When you persist this object using the JPA API, the `activitiesByDayOfWeek` map will be stored in the database with the keys as strings representing the names of the enum constants.

🚀 And that's it! Using enums as persistable map keys is a great way to make your code more readable and maintainable.

Done

✅ Day 16/90 ==> [JavaEE: Day 16/90 - Java Persistence API \(JPA\)](#)

Day 17-18-19-20-21-22-23-24/90

Date : 26-April-2023 *till* 3-May-2023

No progress since I had 2 exams one after another. ("Linear Algebra" & "The theory of Languages and machines")

Day 25/90

Date : 04-May-2023

REVISION

Done

✅ Day 25/90

Day 26/90

Date : 05-May-2023

Java Persistence API (JPA) (continue)

Keying Persistable Maps by *Basic Type*

 **Note:** Keying Persistable Maps by Basic Type in JPA JavaEE with `@OneToMany` and `@MapKey`

For example, let's say we have an `Employee` entity and a `Department` entity. We want to create a map that associates each employee with their department. To do this, we can annotate the `Employee` entity with the `@OneToMany` annotation and set the target entity to the `Department` entity using the `mappedBy` attribute.

```
@Entity
public class Employee {
    @Id
    private Long id;
    // other attributes

    @ManyToOne
    private Department department;

    // getters and setters
}

@Entity
public class Department {
    @Id
    private Long id;
    // other attributes

    @OneToMany(mappedBy = "department")
    @MapKey(name = "id")
    private Map<Long, Employee> employeesByDepartment;

    // getters and setters
}
```

Here, we're using the `@MapKey` annotation to specify that we want the key of the map to be the id of the `Employee` entity. This means that we can access the employees for a given department using its id.

When we run this code, JPA will generate two tables in the database: one for `Employee` and one for `Department`. The `Department` table will have a foreign key column referencing the primary key of the `Employee` table.

=> when we persist data using this mapping, it will create three tables in the database:

1. `Employee` table: This table will have columns for all attributes in the `Employee` class, including a foreign key column referencing the primary key of the associated `Department`.
2. `Department` table: This table will have columns for all attributes in the `Department` class, as well as a primary key column.

3. `Department_employee` table: This table will be generated automatically by Hibernate to manage the one-to-many relationship between `Department` and `Employee`. It will have two columns: a foreign key column referencing the primary key of the `Department` table, and a foreign key column referencing the primary key of the `Employee` table.

Here's some example data :

```
-- Department table
| id | name      |
|----|-----|
| 1  | Sales     |
| 2  | Marketing |
| 3  | IT        |

-- Employee table
| id | name    | department_id |
|----|-----|-----|
| 1  | Alice   | 1           |
| 2  | Bob     | 1           |
| 3  | Charlie | 2           |
| 4  | Dave    | 3           |

-- Department_employee table
| department_id | employee_id |
|-----|-----|
| 1          | 1          |
| 1          | 2          |
| 2          | 3          |
| 3          | 4          |
```

In this example, there are three departments (`Sales`, `Marketing`, and `IT`) and four employees (`Alice`, `Bob`, `Charlie`, and `Dave`). The `department_id` column in the `Employee` table is used to link each employee to a specific department, and the `Department_employee` table is used to manage the relationship between departments and employees.

Keying Persistable Maps by *Entities*

Let me explain how to key persistable maps by entity in JPA JavaEE with an example 🤖

To start, let's say we have two entities `Employee` and `Department` in our database. We want to specify the ranks of employees by integer value. The higher the value, the lower the rank of employees. To do this, we can create a map called "`employeeRanks`" in the `Department` entity using the `@OneToMany` annotation.

🔑 Keying Persistable Maps by Entity:

```
@Entity
public class Department {
    @Id
    private Long id;
```

```

    @OneToMany(mappedBy = "department")
    @MapKeyJoinColumn(name="employee_id")
    private Map<Employee, Integer> employeeRanks;

    // getters and setters
}

@Entity
public class Employee {
    @Id
    private Long id;

    @ManyToOne
    private Department department;

    // other fields, getters and setters
}

```

In the `Department` entity, we use the `@OneToMany` annotation with the `"mappedBy"` attribute to indicate that this relationship is bidirectional. This means that changes made to the `"employeeRanks"` map on the Department side will be updated in the Employee entity as well.


We also use the `@MapKeyJoinColumn` annotation to specify that the key of the map should be the Employee entity, with the name of the join column being `"employee_id"`.

For the `Employee` entity, we use the `@ManyToOne` annotation to establish the relationship between the Employee and Department entities.

 **Database Tables:** This setup will create three tables in your database:

- Department (with columns: id)
- Employee (with columns: id, department_id)
- Department_Employee (with columns: department_id, employee_id, rank)

The Department_Employee table serves as the join table between the Department and Employee entities, with an additional "rank" column to store the employee rank.

 **Example:** Let's say we have a Department entity with id=1, and two Employee entities with ids 2 and 3 respectively. We want to set their ranks as follows:

- Employee with id 2 has rank 2
- Employee with id 3 has rank 1

Department--table

id
1

Employee--table

id	department_id
2	1
3	1

Department_Employee--table

department_id	employee_id	rank
1	2	2
1	3	1









As you can see, the Department table has one row with id=1. The Employee table has two rows with ids 2 and 3 respectively, both of which have department_id=1 to indicate that they belong to the same department.



































The Department_Employee table serves as the join table between the Department and Employee entities. In this case, it has two rows with department_id=1, indicating that both employees belong to the same department. The "employee_id" column identifies which Employee entity each row corresponds to (2 or 3), and the "rank" column stores their respective ranks (2 for Employee id 2, and 1 for Employee id 3).

 **Note :** In a persistable map, When

- the "Key" is a *basicType* (Integer, String, etc), and the "Value" is an *entity*, then we are bound to put `Any_Relationship_Annotation` i.e. (`@OneToMany` etc).
- the "Value" is a *basicType*, then we are bound to put `@ElementCollection`. It doesn't matter what is the "Key" in this case.

Summary

- Java Persistence API (JPA) : A framework for managing relational data in Java applications.
- Setting up Payara Server : Steps to install and configure the Payara Server, which is a popular application server that supports JPA.
- JPA Entity : An annotated Java class that represents a persistent object in a database.
- Customizing Table Mapping : Techniques for mapping an entity to a specific table in the database.
- Using Super Classes : Inheriting properties from a parent class when defining JPA entities.
- Overriding Super Class Field : Changing the behavior of inherited fields in a subclass.
- Mapping Simple Java Types : Translating basic Java data types (such as int and String) to their equivalent database types.
- Transient Fields : Fields that are not persisted to the database.

- Field Access Type : Defining whether JPA should access fields directly or through getter/setter methods.
- Mapping Enumerator Type : Persisting Java enums as database values.
- Mapping Large Objects (e.g. images) : Techniques for storing large binary data (such as images) in a database.
- Lazy & Eager Fetching Of Entity State  : Specifying when JPA should load related objects eagerly (right away) or lazily (on demand).
- Mapping Java 8 DateTime Types : Storing date/time data using the new Java 8 Date/Time API.
- Mapping Embeddable classes : Including non-entity classes inside an entity class.
- Mapping Primary Keys : Defining how JPA should generate primary keys for entities.
- Auto Primary Key Generation Strategy : Configuring JPA to automatically generate primary keys for entities.
- Entity Relationship Mapping : Defining how different JPA entities are related to each other.
- Roles  : Identifying the role that an entity plays in a relationship (such as "owner" or "child").
- Directionality \leftrightarrow : Specifying whether a relationship is unidirectional or bidirectional.
- Cardinality , : Defining the number of entities that can be associated with another entity in a relationship.
- Ordinality : Defining the order of entities in a relationship.
- Ownership of Relationships : Identifying which entity "owns" the relationship and controls cascade operations (such as deletes).
- Unidirectional : A relationship where one entity has a reference to another entity, but the reverse is not true.
- Bidirectional \leftrightarrow : A relationship where two entities have references to each other.
- @ManyToOne : An annotation used to define a many-to-one relationship between two entities.
- Fetch Mode  : Specifying how JPA should load related objects (eagerly or lazily) for collection-valued relationships.
- Collection Mapping Of Embeddable Objects and Collection Table  : Techniques for mapping embedded objects and collections to database tables.
- Ordering The Contents Of a Persistable Collection  : Defining the order in which objects should be retrieved from a collection-valued relationship.
- Mapping Persistable Maps  : Techniques for persisting maps (key-value pairs) to a database.
- Using Enums As Persistable Map Keys   : Storing Java enums as keys in a map that is persisted to the database.
- Keying Persistable Maps by Basic Type  : Using basic Java data types (such as String or int) as keys in a map that is persisted to the database.
- Keying Persistable Maps by Entities   : Using other JPA entities as keys in a map that is persisted to the database.

EJB - Enterprise Java Beans

🤖 Imagine you run a restaurant and your customers expect efficient service. You can hire a lot of staff to handle different tasks, but it's hard to manage them all on your own. That's where Enterprise JavaBeans (EJBs) come in!

🍽️ An EJB is like a specialized employee at your restaurant who handles specific tasks, such as taking orders or preparing food. Just like how your employees have specific roles, EJBs have predetermined roles and responsibilities within a JavaEE application.

🚗 EJB stands for Enterprise JavaBeans, which is a technology used in JavaEE to develop distributed and scalable applications.

🏢 In an enterprise application, there are many different components that need to work together, like databases, web servers, and client applications. EJBs provide a standardized way to manage these components by defining roles and responsibilities for each component.

🍷 For example, imagine you want to add a new dish to your menu. With EJBs, you can easily create a "Recipe EJB" that manages the data around your new dish, like its ingredients, cooking instructions, and nutritional information. This makes it easy for other parts of your application, like the ordering system, to access and use this information.

💻 Some examples of EJBs in JavaEE include:

- Entity Beans: Used for representing persistent data in a database.
- Message-Driven Beans: Used for processing messages asynchronously.
- 🔍 For example, Session Beans are EJBs that handle business logic and are responsible for processing client requests. Entity Beans represent persistent data stored in a database, while Message-Driven Beans handle asynchronous messaging.
- 📦 EJBs also provide built-in services like transaction management, security, and persistence, which makes it easier for developers to focus on writing business logic rather than worrying about low-level details.
- 📈 One of the big advantages of using EJBs is that they make it easy to scale your application horizontally (i.e., across multiple machines) as demand grows. Because each EJB has a well-defined role and can communicate with other EJBs, you can add more instances of an EJB to handle increased load without disrupting the rest of your application.
- 🔒 Another advantage of EJBs is that they make it easier to secure your application. By defining roles and permissions for each EJB, you can control access to sensitive data and functionality.

🚗 EJBs are a technology in JavaEE that allow developers to build scalable, maintainable, and distributable applications. They provide a set of services, such as transaction management, security, and persistence, that help simplify the development process.

Features of EJB

1. Declarative Metadata

Declarative metadata is information about your code that is specified outside of the code itself. In the context of Java EE EJB, this refers to annotations or XML files that provide additional information about your Enterprise JavaBeans. This metadata helps the application server understand how to manage and run your EJB.

2. Configuration by Exception

Configuration by exception is a design pattern used in Java EE EJB where default settings are used unless otherwise specified. This means that instead of having to explicitly configure every aspect of your EJB, you only need to specify any exceptions or deviations from the default behavior.

3. Dependency Management

Dependency management is the process of identifying and resolving dependencies between different components in your application. In Java EE EJB, this involves managing dependencies between your Enterprise JavaBeans and other resources like databases, message queues, and web services.

4. Lifecycle Management

Lifecycle management refers to the various stages an EJB goes through during its lifetime, including creation, activation, passivation, and removal. The application server manages the lifecycle of EJBs and ensures they are instantiated and destroyed at the appropriate times.

5. Scalability

Scalability refers to the ability of an application to handle increasing amounts of traffic or workload. In Java EE EJB, this involves horizontal scaling, where multiple instances of an EJB are created and distributed across multiple servers to handle increased demand.

6. Transactionality

Transactionality refers to the ability of an EJB to participate in transactions - groups of operations that are either all completed successfully or rolled back if any one operation fails. Transactions ensure data consistency and integrity in complex systems.



7. Security

Security refers to the measures taken to protect an application from unauthorized access and ensure data confidentiality, integrity, and availability. In Java EE EJB, this involves authentication (verifying user identities) and authorization (granting permissions based on those identities).

8. Portability

Portability refers to the ability to move an application between different environments or platforms without modification. In Java EE EJB, this means that your Enterprise JavaBeans should be able to run on any Java EE-compliant application server without needing to be modified.

Architecture of EJB

 Note: Different Kinds of EJB Architecture 



Enterprise JavaBeans (EJB) is a technology used to develop modular components that can be distributed across different systems and platforms. There are three types of EJB architecture:



1 Session Beans :

Session beans are used to represent a transient conversation between a client and an application server. They can be stateless or stateful, depending on whether or not they maintain state information between method invocations.

Stateless session beans do not maintain conversational state, while *stateful* session beans maintain state information. The main advantage of session beans is that they provide a way to encapsulate business logic in a module that can be accessed by multiple clients. These are the most common type of EJB. They represent individual client sessions and perform business logic. For example, a session bean might handle user authentication or manage a shopping cart for an online store.

====> *Stateful session* :

Here's an example: Let's say you have a shopping cart application and you want to keep track of the items that a particular user has added to their cart. You can use a stateful EJB to maintain the state of the user's shopping cart throughout their session.  

When the user adds an item to their cart, the stateful EJB updates its state accordingly. When the user removes an item from their cart, the EJB updates its state again. This way, the EJB can keep track of the user's shopping cart throughout their session.  

```
@Stateful
public class ShoppingCart implements ShoppingCartRemote {

    private List<String> items = new ArrayList<>();

    public void addItem(String item) {
        items.add(item);
    }

    public void removeItem(String item) {
        items.remove(item);
    }

    public List<String> getItems() {
        return items;
    }

    @Remove
    public void checkout() {
        // Perform checkout logic here
    }
}
```

In this example, we define a stateful EJB called "ShoppingCart" that implements the remote interface "ShoppingCartRemote". The stateful nature of this EJB is evident by the fact that it maintains a list of items added to the shopping cart.

The `addItem` and `removeItem` methods allow the client to add or remove items from their shopping cart, while the `getItems` method allows the client to retrieve the current contents of their cart.

Finally, the `checkout` method performs any necessary actions when the client checks out their shopping cart, such as calculating the total cost and updating the inventory. The `@Remove` annotation indicates that the EJB should be removed once the client has checked out.

====> *Stateless session* :

Here's an example: Let's say you have a banking application that exposes a service to transfer money from one account to another. You can use a stateless EJB to implement this service. The EJB would handle the transfer operation, but it wouldn't maintain any state about the transfer itself. 🔍 💡

When a client submits a transfer request, the stateless EJB performs the necessary operations to transfer the money and returns a result. The EJB doesn't need to keep track of the transfer beyond that point, as it has no further impact on the operation of the system. 📈 🏦

One advantage of using stateless EJBs is that they can be more scalable than stateful EJBs, as they don't require resources to maintain conversational state with clients. However, it's important to carefully consider whether a stateless EJB is appropriate for your application, as they may not be suitable for all use cases. 🤖

Here's an example of a stateless EJB in Java code:

```
@Stateless
public class BankTransfer implements BankTransferRemote {

    public boolean transfer(String accountFrom, String accountTo, double amount) {
        // Perform the necessary operations to transfer the money
        return true;
    }
}
```

In this example, we define a stateless EJB called "BankTransfer" that implements the remote interface "BankTransferRemote".

The `transfer` method takes three parameters: the account to transfer from, the account to transfer to, and the amount to transfer. When invoked, the EJB performs the necessary operations to transfer the money between the accounts and returns a result indicating whether the transfer was successful or not.

2 Singleton Beans :

A singleton bean in EJB is a type of EJB that only allows one instance to be created and shared across multiple clients. The singleton bean is instantiated when the application server starts up and remains in memory until the application server shuts down. 👥 🏠 💬

Here's an example:

```

@Singleton
public class MySingletonBean {

    private int count = 0;

    public int getCount() {
        return count;
    }

    public void incrementCount() {
        count++;
    }
}

```

In this example, we have defined a singleton bean called `MySingletonBean`, which has a single instance that is shared by all clients. The bean has a method `getCount()` that returns the current value of a private variable `count`, and a method `incrementCount()` that increments the value of `count` by one. 🙌 💻 🔥

To use this singleton bean in your EJB application, you would simply inject it into your client code like this:

```

@Stateless
public class MyStatelessBean {

    @EJB
    private MySingletonBean singletonBean;

    public void doSomething() {
        int count = singletonBean.getCount();
        singletonBean.incrementCount();
        // do something with count
    }
}

```

In this example, we have a stateless bean called `MyStatelessBean` that injects an instance of `MySingletonBean` using the `@EJB` annotation. The stateless bean then calls the `getCount()` method to get the current value of `count`, and the `incrementCount()` method to increment it.

3 Message-Driven Beans :

Message-driven beans (MDBs) are used to process messages asynchronously. They are triggered by messages sent to a message queue or topic, and can perform processing tasks based on the contents of the message. MDBs are typically used in enterprise integration scenarios, where disparate systems need to exchange data asynchronously. One of the key advantages of MDBs is that they allow for loosely coupled integrations between systems. These beans receive and process messages asynchronously. They are commonly used in messaging systems and event-driven architectures. For example, a message-driven bean might process incoming orders for an online store or update a user's account based on a system event.

Lifecycle of EJBs

1. Stateful Session Beans 🌱❤️

Stateful Session Beans (SFSB) maintain state information between client invocations, meaning that each client has a unique bean instance. The lifecycle of an SFSB is as follows:

- **Creation:** The container creates a new instance of the SFSB when a client makes a request.
- **Method Invocation:** The client invokes methods on the bean, which modifies its state.
- **Passivation:** If the bean is not accessed for a specified amount of time, the container can choose to serialize its state and remove it from memory to free up resources.
- **Activation:** When a client requests a passivated SFSB, the container restores its state from the serialized data and activates the bean instance.
- **Removal:** The container removes the SFSB instance when the client session ends or when the bean is explicitly removed by the client.

Example: A shopping cart in an online store would be a good example of an SFSB since it needs to maintain state information (items in the cart) for a specific client session.

2. Stateless Session Beans 🌱🔄

Stateless Session Beans (SLSB) do not maintain any state information between client invocations, meaning that each client request is processed independently. The lifecycle of an SLSB is as follows:

- **Creation:** The container creates a pool of bean instances when the application starts up.
- **Method Invocation:** The client invokes methods on the bean, which performs some processing and returns a result.
- **Destruction:** The container returns the bean instance to the pool after the method invocation, making it available for another client request.

Example: A calculator that performs some computation based on the input values would be a good example of an SLSB since it does not need to maintain any state information between client invocations.

3. Singleton Session Beans 🌱👑

Singleton Session Beans (SSB) maintain a single instance of the bean for the entire application, meaning that all clients share the same instance. The lifecycle of an SSB is as follows:

- **Creation:** The container creates a single instance of the SSB when the application starts up.
- **Method Invocation:** All clients invoke methods on the same instance, which modifies its state.
- **Destruction:** The container destroys the SSB instance when the application shuts down or when it is explicitly removed by the client.

Example: A configuration manager that maintains some global settings for the entire application would be a good example of an SSB since it needs to maintain a single instance with the same state information across all client requests.

Done

Day 27/90

Date : 06-May-2023

EJB - Enterprise Java Beans (continue)

Transaction

🤖 So what is a transaction? A transaction is a logical unit of work that groups together several database operations. These operations are executed as a single unit, which either succeeds or fails completely.

💰 Let's say you want to transfer money from one bank account to another. A transaction would ensure that both accounts are updated correctly, and that no money is lost along the way. If something goes wrong during the transfer (e.g. a network error), the transaction can be rolled back, and both accounts will remain unchanged.

📄 In JavaEE, transactions are managed by the container (e.g. Tomcat, Glassfish, JBoss). You start a transaction by marking a method with the `@Transactional` annotation. The container then takes care of managing the transaction for you.

🚀 Here's an example:

```
@Transactional
public void transferMoney(Account sourceAccount, Account destinationAccount, double
amount) {
    try {
        // Deduct the amount from the source account
        sourceAccount.setBalance(sourceAccount.getBalance() - amount);
        accountDao.update(sourceAccount);

        // Add the amount to the destination account
        destinationAccount.setBalance(destinationAccount.getBalance() + amount);
        accountDao.update(destinationAccount);
    } catch (Exception e) {
        // Something went wrong, so roll back the transaction
        throw new RuntimeException(e);
    }
}
```

👁 In this example, we're transferring money from one account to another. We've marked the method with the `@Transactional` annotation, which tells the container to manage the transaction for us.

🏢 First, we deduct the amount from the source account and update its balance in the database. Then, we add the same amount to the destination account and update its balance as well. If anything goes wrong during the transfer (e.g. an exception is thrown), the container will automatically roll back the transaction, and both accounts will remain unchanged.

👍 And that's transactions in a nutshell! They're a powerful tool for ensuring data integrity and consistency in your applications.

Properties Of Transaction

1. 🔥 **Atomicity** : Atomicity is the property of a transaction that ensures that all the database operations within the transaction are treated as a single, indivisible unit. This means that either all the operations within the transaction succeed or none of them do. If any operation fails, the entire transaction is rolled back to the previous state.

💎 For example, let's say you're buying a pair of shoes online. The transaction would involve deducting the amount from your bank account and updating the inventory system to reduce the number of shoes in stock. If either operation fails, the entire transaction must fail, and the bank account and inventory should remain unchanged.

2. 🧑 **Consistency** : Consistency is the property of a transaction that ensures that the database remains in a valid state throughout the entire transaction. This means that any changes made to the database within the transaction must satisfy all the integrity constraints and business rules defined for the database.

🛒 For example, if you're buying a product online, the transaction must ensure that the total cost of the purchase is correctly calculated and that any discounts or taxes are applied correctly according to the business rules.

3. 💛 **Isolation** : Isolation is the property of a transaction that ensures that each transaction is independent of other concurrent transactions executing on the same database. This means that the results of one transaction should not affect the results of other transactions executing concurrently.

👥 For example, if two people are booking tickets for the same movie at the same time, each transaction should be isolated from the other. This ensures that one person's booking does not affect the other person's booking, and both bookings can proceed independently.

4. 💾 **Durability** : Durability is the property of a transaction that ensures that once the transaction is committed, its effects are permanent and persistent, even in the face of system failure (such as power outage or hardware failure). This means that the changes made to the database within a committed transaction must survive any subsequent system failures.

👉 For example, if you're transferring money from one account to another, the transaction must ensure that the transfer is recorded permanently, even if there's a power outage or hardware failure during the transaction. Once the transaction is committed, it should be durable, and the money transfer should be reflected in both accounts, even in the event of a system failure.

Transaction Management Attributes

In JavaEE, transaction management is a way of ensuring that changes made to a database or other data store are handled consistently and reliably. It's like making sure that each step in a process is completed before moving on to the next one.

To understand this better, imagine you're trying to transfer money from one bank account to another. You wouldn't want the money to be deducted from one account without being added to the other account, right? That's where transaction management comes in.

JavaEE has a few different attributes for managing transactions, but let's focus on two of the most basic ones: `REQUIRED` and `REQUIRES_NEW`.

- The `REQUIRED` attribute means that the method being called must participate in an existing transaction if one exists. If there is no current transaction, a new one will be started.

For example, imagine you have a method that updates the balance of a bank account:

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void updateAccountBalance(String accountNumber, double amount) {
    // code to update the balance goes here
}
```

If this method is called while a transaction is already in progress (e.g. because it's part of a larger banking transaction), it will participate in that transaction. If not, a new transaction will be started specifically for this method.

- The `REQUIRES_NEW` attribute means that a new independent transaction will always be started, even if there is already an existing transaction.

Using the same example as above, here's how you could modify the code to use `REQUIRES_NEW`:

```
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void updateAccountBalance(String accountNumber, double amount) {
    // code to update the balance goes here
}
```

In this case, a new transaction will always be started whenever this method is called, regardless of whether there is already a transaction in progress.

- `MANDATORY`: This attribute specifies that the method being called must be executed within an active transaction context. If there is no active transaction, a `TransactionRequiredException` will be thrown.

For example:

```
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public void updateUserData(String username, String data) {
    // code to update user data goes here
}
```

This method can be called only if there is already an active transaction in progress. If there is no active transaction, then a `TransactionRequiredException` will be thrown.

- **SUPPORTS**: This attribute specifies that the method being called supports transactions, but does not require one. If there is an active transaction in progress, the method will participate in it. If there is no active transaction, the method will execute without a transaction.

For example:

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public int getUserCount() {
    // code to count users in the database goes here
}
```

This method can be called with or without an active transaction in progress. If there is an active transaction, it will participate in that transaction. If not, it will execute without a transaction.


- **NOT_SUPPORTED**: This attribute specifies that the method being called should not be executed within a transaction context. If there is an active transaction in progress, it will be suspended while this method executes.


For example:

```
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public List<String> getPublicData() {
    // code to fetch public data goes here
}
```

This method should never be called within an active transaction. If there is an active transaction in progress, it will be suspended while this method executes. When the method completes, the transaction will resume.


Persistence Unit vs Persistence Context - Intro

 A Persistence Unit in Java EE is like a container for all the information needed to connect to a database. It contains the configuration and mapping files, as well as the database connection details.

 A Persistence Context in Java EE is like a workspace where data is manipulated before it is saved to the database. It acts as a cache of entity instances that are managed by the EntityManager.

In simpler terms, a Persistence Unit provides the information needed to connect to a database, while a Persistence Context manages the interaction between the application and the database by caching and manipulating data.

Entity Manager - How To Get Access

 What is an Entity Manager? An Entity Manager is a Java EE component that manages the lifecycle of entities within a Java Persistence API (JPA) context. In simpler terms, it's what allows you to interact with your database using JPA.

👉 How do you get access to an Entity Manager? You can obtain an instance of the Entity Manager by using Dependency Injection or JNDI lookup. Here's an example of using DI:

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

In this example, we're injecting an instance of the Entity Manager into a managed bean using the `@PersistenceContext` annotation.

👉 How do you use the Entity Manager? Once you have access to the Entity Manager, you can use it to perform CRUD (Create, Read, Update, Delete) operations on your entities. Here's an example of persisting an entity to the database:

```
Customer customer = new Customer();
customer.setName("John Doe");
customer.setEmail("johndoe@example.com");

entityManager.persist(customer);
```

In this example, we're creating a new `Customer` entity, setting some properties on it, and then using the `persist()` method of the Entity Manager to save it to the database.

👉 How does this compare to Spring Framework? In Spring Framework, you can also use JPA for database interaction, but the way you obtain an instance of the Entity Manager is slightly different. Instead of using Dependency Injection or JNDI lookup, you would typically use the `LocalContainerEntityManagerFactoryBean` to create an instance of the Entity Manager.

Here's an example of configuring the Entity Manager in Spring Framework:

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean em =
        new LocalContainerEntityManagerFactoryBean();
    em.setDataSource(dataSource());
    em.setPackagesToScan(new String[] { "com.example.entities" });

    JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    em.setJpaVendorAdapter(vendorAdapter);

    return em;
}
```

In this example, we're creating a `LocalContainerEntityManagerFactoryBean` and configuring it with a datasource, packages to scan for entities, and a JPA vendor adapter (in this case, Hibernate).

Once you have configured the Entity Manager in Spring Framework, you can use it in a similar way to Java EE:

```

@Autowired
private EntityManager entityManager;

@Transactional
public void saveCustomer(Customer customer) {
    entityManager.persist(customer);
}

```

In this example, we're injecting an instance of the Entity Manager using `@Autowired` and then using it to persist a `Customer` entity to the database.

Entity Manager - Operations

Let's consider two entities `Student` and `Course` with a many-to-many relationship between them. One student can enroll in multiple courses, and one course can have multiple students enrolled.

Here are the entity classes with their respective relationships:

```

@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany(mappedBy = "students")
    private List<Course> courses = new ArrayList<>();

    // constructors, getters, setters, and other methods
}

@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany
    @JoinTable(
        name = "course_student",
        joinColumns = @JoinColumn(name = "course_id"),
        inverseJoinColumns = @JoinColumn(name = "student_id"))
    private List<Student> students = new ArrayList<>();

    // constructors, getters, setters, and other methods
}

```

Now, let's perform some entity manager operations on these entities:

1. `persist`: This operation is used to save a new entity to the database.

👉 Example: Let's create a new `Student` entity and enroll them in an existing `Course` entity using the `persist` operation:

```
EntityManager entityManager = getEntityManager(); // assume this method returns an
instance of EntityManager

// Retrieve an existing course from the database
Course course = entityManager.find(Course.class, courseId);

// Create a new student
Student student = new Student();
student.setName("John");

// Enroll the student in the course
course.getStudents().add(student);

// Save the new student to the database
entityManager.persist(student);
```

2. `find`: This operation is used to retrieve an entity from the database based on its primary key.

👉 Example: Let's retrieve a specific `Course` entity from the database and print its enrolled students using the `find` operation:

```
EntityManager entityManager = getEntityManager(); // assume this method returns an
instance of EntityManager

// Retrieve the course with ID 1
Course course = entityManager.find(Course.class, 1L);

// Print the names of all students enrolled in the course
for (Student student : course.getStudents()) {
    System.out.println(student.getName());
}
```

3. `remove`: This operation is used to delete an entity from the database.

👉 Example: Let's remove a specific `Student` entity from the database along with their enrollment in all courses using the `remove` operation:

```

EntityManager entityManager = getEntityManager(); // assume this method returns an
instance of EntityManager

// Retrieve the student we want to delete
Student student = entityManager.find(Student.class, studentId);

// Remove the student from all courses they are enrolled in
for (Course course : student.getCourses()) {
    course.getStudents().remove(student);
}

// Remove the student itself
entityManager.remove(student);

```

4. `merge`: This operation is used to update an existing entity in the database.

👉 Example: Let's update a specific `Course` entity in the database by adding a new `Student` entity to it using the `merge` operation:

```

EntityManager entityManager = getEntityManager(); // assume this method returns an
instance of EntityManager

// Retrieve the course we want to update
Course course = entityManager.find(Course.class, courseId);

// Create a new student
Student student = new Student();
student.setName("Jane");

// Enroll the student in the course
course.getStudents().add(student);

// Update the course in the database
entityManager.merge(course);

```

Cascade Operations

Cascade operations refer to a set of actions that are automatically applied to related entities when an operation is performed on a parent entity. These operations include persisting (📄), removing (✖), refreshing (🔄), merging (🔗), detaching (👤), or applying the operation to all related entities (💡).

For example, let's say we have a database with two tables: `Order` and `Item`. An order can have multiple items associated with it. We can use cascade operations to automatically perform actions on the associated items when an action is performed on the order.

If we set the cascade type to `PERSIST`, then when we save a new order to the database, any new items associated with that order will also be saved automatically. Similarly, if we set the cascade type to `REMOVE`, then when we delete an order from the database, any associated items will also be deleted automatically.

Here's some example code to demonstrate :

1. Persist (📁): This operation is used to save new entities into the database.

```
@Entity
public class Order {
    @Id
    private Long id;

    @OneToMany(mappedBy="order", cascade=CascadeType.PERSIST)
    private List<Item> items;

    // getters and setters
}

@Entity
public class Item {
    @Id
    private Long id;

    @ManyToOne
    private Order order;

    // getters and setters
}

// create a new order with two items
Order order = new Order();
Item item1 = new Item();
Item item2 = new Item();

// associate the items with the order
item1.setOrder(order);
item2.setOrder(order);

// add the items to the order's list of items
order.getItems().add(item1);
order.getItems().add(item2);

// persist the order (and its associated items)
entityManager.persist(order);
```

In this example, we create a new `Order` object and associate two new `Item` objects with it. We then set the cascade type to `PERSIST` on the `items` field of the `Order` entity, which means that when we persist the `Order` to the database, any new `Item` objects associated with it will also be persisted automatically.

2. Remove (✖): This operation is used to delete existing entities from the database.


```
// remove an order (and its associated items)
Order order = entityManager.find(Order.class, orderId);
entityManager.remove(order);
```

In this example, we retrieve an existing `Order` object from the database using its `id` field, and then call `entityManager.remove(order)` to delete it from the database. Because we've set the cascade type to `REMOVE` on the `items` field of the `Order` entity, any associated `Item` objects will also be deleted automatically.

3. Refresh (): This operation is used to reload the state of an entity from the database.

```
// refresh an order (and its associated items)
Order order = entityManager.find(Order.class, orderId);
entityManager.refresh(order);
```


In this example, we retrieve an existing `Order` object from the database using its `id` field, and then call `entityManager.refresh(order)` to reload its state from the database. Because we've set the cascade type to `REFRESH` on the `items` field of the `Order` entity, any associated `Item` objects will also be refreshed automatically.

4. Merge (): This operation is used to update an entity in the database. It merges the state of an entity with the state of the corresponding managed entity in the persistence context.

```
// modify an order (and its associated items) and merge changes into the
// database
Order order = entityManager.find(Order.class, orderId);
order.getItems().remove(0); // remove the first item from the order's list of
// items

entityManager.merge(order); // merge the changes into the database
```

In this example, we retrieve an existing `Order` object from the database using its `id` field, modify it by removing the first `Item` object from its list of items, and then call `entityManager.merge(order)` to merge the changes back into the database. Because we've set the cascade type to `MERGE` on the `items` field of the `Order` entity, any associated `Item` objects will also be merged automatically.

5. Detach (): This operation is used to detach an entity from the persistence context, making it no longer managed.

```
// detach an order (and its associated items) from the persistence context
Order order = entityManager.find(Order.class, orderId);
entityManager.detach(order); // detach the order from the persistence context
```

In this example, we retrieve an existing `Order` object from the database using its `id` field, and then call `entityManager.detach(order)` to detach it from the persistence context. Because we've set the cascade type to `DETACH` on the `items` field of the `Order` entity, any associated `Item` objects will also be detached automatically.

6. All (): This operation applies the specified operation to all related entities.

```
@Entity
public class Order {
    @Id
```

```

private Long id;

@OneToMany(mappedBy="order", cascade=CascadeType.ALL)
private List<Item> items;

// getters and setters
}

// remove an order (and its associated items)
Order order = entityManager.find(Order.class, orderId);
entityManager.remove(order);

```

In this example, we retrieve an existing `order` object from the database using its `id` field, and then call `entityManager.remove(order)` to delete it from.

Entity Detachment

In Java EE (Enterprise Edition), an entity refers to a specific object or data that is managed by a persistence framework like JPA. Entity detachment simply means that the framework stops managing that particular entity object (the entity gets out of the persistence context).

👉 Imagine you have a toy car (🚗) that your friend is playing with. As long as your friend is holding onto the car, they are in control of its movements and can make changes to it. This is similar to how a persistence framework controls an entity object.

👉 Now, if your friend puts the car down (detaches it), it's no longer under their control. They can't move it around anymore or make any changes to it. Similarly, when an entity is detached in Java EE, the persistence framework no longer manages changes to that object.

This can happen for a variety of reasons, such as when an entity is no longer needed in the context of a transaction, or when the persistence context (the set of managed entities) is closed.

Elements Of Persistence Unit

In Java EE, a persistence unit represents a set of entity classes that are managed together by the Java Persistence API (JPA). It provides configuration information to JPA, such as the database connection details and the entity classes to be managed.

Here's what each of the elements means :

```

<persistence-unit name="UserPersistenceUnit" transaction-type="JTA">
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  <jta-data-source>java:/comp/env/jdbc/userDataSource</jta-data-source>
  <class>com.example.User</class>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="create"/>
  </properties>
</persistence-unit>

```

1. The `<persistence-unit>` element is used to define a persistence unit in Java EE. It has several attributes and child elements that are used to configure the persistence unit.

- `name="UserPersistenceUnit"`: This attribute sets the name of the persistence unit to "UserPersistenceUnit". The name is used to uniquely identify the persistence unit within an application.
 - `transaction-type="JTA"`: This attribute specifies the type of transaction management that should be used for this persistence unit. In this case, "JTA" (Java Transaction API) is used, which means that transaction management will be handled by the application server.
2. Persistence provider: 📁 The `<provider>` element is used to specify the JPA provider that should be used to manage the persistence unit. In this case, the Hibernate JPA provider is used (`org.hibernate.jpa.HibernatePersistenceProvider`).
 3. JTA datasource: 🍷 🖥️ The JTA datasource specifies the connection details required to connect to the database. It includes information such as the URL, username, and password. The `<jta-data-source>` element is used to specify the JNDI name of the DataSource that should be used for database connections. In this case, the JNDI name is set to `java:/comp/env/jdbc/userDataSource`.
 4. Entity classes: 🦋 Entity classes are Java classes that represent tables in the database. They typically contain fields or properties that map to the columns in the table. The `<class>` element is used to specify the entity classes that should be managed by the persistence unit. In this case, the entity class `com.example.User` is specified.
 5. Schema & script generation: 📄 ⚙️ Schema and script generation refer to the process of creating database tables from the entity classes. This can be done automatically based on the entity mappings, or manually using SQL scripts.
 6. Properties: ⚙️ The `<properties>` element is used to specify additional properties that should be used when configuring the persistence unit. In this case, the `hibernate.hbm2ddl.auto` property is set to "create", which tells Hibernate to automatically generate the database schema based on the entity mappings.

For example, let's say we have an application that needs to manage a set of user data in a MySQL database. We might define a persistence unit like this in our `persistence.xml` file:

This defines a persistence unit named "UserPersistenceUnit" with a JTA transaction type. The provider is set to use Hibernate, and the JTA datasource is specified as "java:/comp/env/jdbc/userDataSource". We also specify that we want to manage a single entity class called "User". Finally, we set the `hibernate.hbm2ddl.auto` property to "create", which tells Hibernate to automatically generate the database schema based on our entity mappings.

To create this `persistence.xml` file, we can simply create a new XML file in our project and add the above code to it. We then need to make sure that the file is located in the `src/main/resources/META-INF/persistence.xml` directory of our application's classpath.

JPQL - Java Persistence Query Language

JPQL (Java Persistence Query Language) is a powerful tool in the JavaEE world for working with databases 🗄️. It's like a language that allows you to ask questions and retrieve data from your database 💬.

With JPQL, you can write queries to retrieve specific information from your entities, which are like tables in your database 🗄️. You can also join multiple entities together to get more complex results 🧠.

JPQL uses object-oriented terms rather than SQL's table and column terminology 🚀. For example, instead of selecting columns from a table, you select attributes from an entity.

@NamedQuery

In JavaEE, we often use the Java Persistence API (JPA) to work with databases 🗄️. One way to retrieve data from a database using JPA is by writing dynamic queries. These are queries that are built at runtime based on user input 🗄️.

However, there's another approach called "@NamedQuery" that allows you to create pre-defined named queries in your entity classes 📄.

Here's an example of how it works:

```
@Entity
@NamedQuery(name = "findEmployeeByName",
            query = "SELECT e FROM Employee e WHERE e.name = :name")
public class Employee {
    ...
}
```

In this example, we've added a named query to the Employee entity class. The query is named "findEmployeeByName" and selects all employees whose name matches the provided parameter ":name".

we can name the jpql this way too :

```
@Entity
@NamedQuery(name = Employee.FIND_EMPLOYEE_BY_NAME, //<-- this line is updated
            query = "SELECT e FROM Employee e WHERE e.name = :name")
public class Employee {
    private final String FIND_EMPLOYEE_BY_NAME = "Employee.findEmployeeByName";
    //<-- this line is new
    ...
}
```

With this named query, we can then easily execute the query from our code like this:

```
Query query = entityManager.createNamedQuery("findEmployeeByName");
query.setParameter("name", "John Doe");
List<Employee> employees = query.getResultList();
```

or :



```


TypedQuery<Employee> query =
entityManager.createNamedQuery(Employee.FIND_EMPLOYEE_BY_NAME, Employee.class);
List<Employee> employees = query.getResultList();

// we can make it simpler :
List<Employee> employees =
entityManager.createNamedQuery(Employee.FIND_EMPLOYEE_BY_NAME,
Employee.class).getResultList();

```

This will fetch all the employees with the name "John Doe" from the database.

The advantage of using named queries over dynamic queries is that they are pre-compiled and can be reused multiple times without having to rebuild the query each time the query is executed . This can lead to better performance and less overhead in your application .

When comparing Spring Framework to JavaEE, both frameworks support named queries and dynamic queries. However, Spring provides more advanced features, such as the ability to create specifications and criteria queries, which allows for even more flexibility and reusability in your codebase .

In terms of using named queries in Spring Boot, the process is as following : You would define your named query in the entity class :

```

@Entity
@NamedQuery(name = "findEmployeeByName",
            query = "SELECT e FROM Employee e WHERE e.name = :name")
public class Employee {
    ...
}

```



And then you would use the EntityManager or a JpaRepository to execute the named query :

```

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
    List<Employee> findEmployeesByName(String name);
}

```

In this example, we're using the JpaRepository interface provided by Spring Data JPA to automatically create a repository implementation for us. We define the "findEmployeesByName" method just like before, and Spring Data JPA takes care of generating the query for us based on the method name and signature.

The advantage of using named queries in Spring Boot is the same as in Spring Framework and JavaEE: they are pre-compiled and can be reused multiple times without having to rebuild the query each time the query is executed . This can lead to better performance and less overhead in your application .

Combined Path Expressions

In JPQL, you can use combined path expressions to navigate through multiple related entities in a single query. This is useful when you need to retrieve data from associated entities that are linked by relationships like One-to-One, One-to-Many, or Many-to-Many.

For example, let's say we have two entities: `Order` and `Product`. An order can have many products, so we have a One-to-Many relationship between `Order` and `Product`. We want to retrieve the product name and price for all products belonging to a specific order.

Here's how we can do it using combined path expressions in JPQL :

```
// Define the JPQL query string with combined path expression
@NamedQuery(name = Price.GET_ORDER_NAME_AND_PRICE, query = "SELECT p.name, p.price
FROM Price p");
...

// Execute the query and retrieve the result as a list of Object[] arrays
...
public Collection<Object[]> getOrderNameAndPrice() {
    return entityManager.createQuery(Price.GET_ORDER_NAME_AND_PRICE,
    object[].class).getResultList();
}
```

Done

☒ Day 27/90 ==> [JavaEE: Day 27/90 - EJB \(Enterprise Java Beans\)](#)

Day 28/90

Date : 08-May-2023

JPQL - Java Persistence Query Language (continue)

Constructor Expression

🤖 First, let's define what constructor expressions are. In JPQL, constructor expressions allow you to select specific fields from your entities and map them to a custom class or interface that you define. This can be useful when you want to retrieve only certain data from your database and store it in a custom object.

💡 So, when should you use constructor expressions? You might use them when you want to:

- Retrieve a subset of data from your database
- Map this data to a custom object or interface

- Use this custom object or interface in other parts of your application

👤 Let's use an example to illustrate this. Suppose we have two entities: Employee and Department. Each Employee works for one Department, and each Department has many Employees. We also have a Parking entity, which has a one-to-one relationship with Employee (each Employee has a parking spot).

📄 Here's an example of a named query that uses a constructor expression to select some fields from the Employee and Department entities and map them to a custom object called EmployeeDto:

```
@NamedQuery(  
    name = Employee.findEmployeeswithDeptAndParking,  
    query = "SELECT NEW com.example.EmployeeDto(e.name, d.name, p.location) FROM  
Employee e JOIN e.department d LEFT JOIN e.parking p WHERE e.salary >  
:salaryThreshold ORDER BY e.name"  
)
```

🧐 Let's break this down. The `SELECT` clause specifies that we want to create a new EmployeeDto object, which has three constructor arguments: `e.name`, `d.name`, and `p.location`. These correspond to the name of the Employee, the name of their Department, and the location of their Parking spot.

💛 The `FROM` clause specifies that we want to join the Employee entity with its associated Department, and optionally left join with Parking. We also add a WHERE clause to filter out Employees whose salary is below a certain threshold, and an `ORDER BY` clause to sort the results by name.

🎉 Now we can use this named query in our code to retrieve a list of EmployeeDto objects that contain only the data we need!

From Clause - Join

🔍 The `FROM` clause is used in JPQL (Java Persistence Query Language) to specify one or more entity types that will be included in the query.

💛 When you want to include multiple entities in your query, you can use the `JOIN` keyword to join them together based on a relationship between the entities.

👉 Here's an example of using the `FROM` clause with a join:

Let's say we have two entities, `Employee` and `Department`, with a relationship between them where each employee belongs to one department. We want to retrieve a list of all employees along with their department names.

```
SELECT e.name, d.name  
FROM Employee e  
JOIN e.department d
```

In this query, we are selecting the `name` property from both the `Employee` and `Department` entities. We are joining the `Employee` and `Department` entities together using the `JOIN` keyword, and specifying that we want to join on the `department` property of the `Employee` entity.

🤖 Here's how to interpret the query:

- Start by selecting all `Employee` objects (`e`) and their associated `Department` objects (`d`).
- Then, join the `Employee` and `Department` objects together using the `JOIN` keyword, and specify that you want to join on the `department` property of the `Employee` entity.
- Finally, select the `name` properties from both entities.

🚀 The result of running this query would be a list containing pairs of employee name and department name values.

you can use the `FROM` clause with joins in a JPA named query using the `@NamedQuery` annotation. Here's an example:

Let's say we want to define a named query to retrieve all employees along with their department names, sorted by department name in ascending order. We can define the named query like this:

```
@NamedQuery(  
    name = Employee.findEmployeeswithDepartment,  
    query = "SELECT e.name, d.name FROM Employee e JOIN e.department d ORDER BY  
            d.name ASC"  
)
```

In this named query, we are selecting the `name` property from both the `Employee` and `Department` entities. We are joining the `Employee` and `Department` entities together using the `JOIN` keyword, and then ordering the results by the `name` property of the `Department` entity in ascending order.

To execute this named query, you can use the `createNamedQuery` method of the entity manager:

```
TypedQuery<Object[]> query =  
    em.createNamedQuery(Employee.findEmployeeswithDepartment, Object[].class);  
List<Object[]> results = query.getResultList();
```

In this example, we are creating a typed query that will return an array of objects containing the employee name and department name for each result. The `getResultList` method is used to execute the query and retrieve the results.

From Clause - Join Maps

So, JPQL stands for Java Persistence Query Language and it's used to write queries against entities and their persistent state. The "from clause" is one of the most important clauses in a JPQL query, as it specifies the entity or entities to be queried.

Now, when we talk about "join maps," what we're really referring to is a join between two entities where one of the entities has a map attribute. This can be a bit tricky to understand at first, but let me give you an example.

Let's say we have two entities: Customer and Order. Each customer can have multiple orders, so we represent this relationship using a map in the Customer entity:

```

@Entity
public class Customer {
    @Id
    private Long id;

    @OneToMany(mappedBy = "customer")
    private Map<Long, Order> orders;

    // getters and setters
}

```

In this example, the "orders" attribute is a Map where the keys are order IDs and the values are Order objects.

Now, let's say we want to write a JPQL query that joins the Customer and Order entities on their respective IDs. We can do this using the following syntax:

```

SELECT c, o FROM Customer c JOIN c.orders o WHERE c.id = :customerId AND o.id = :orderId

```

In this example, we're selecting both the Customer and Order entities (hence the "c, o" in the SELECT clause), joining them on the "orders" attribute of the Customer entity, and filtering the results based on the IDs of both entities.

We can also use @NamedQuery annotation to define this query in our entity class like this:

```

@Entity
@NamedQuery(
    name = "Customer.findOrder",
    query = "SELECT c, o FROM Customer c JOIN c.orders o WHERE c.id = :customerId AND o.id = :orderId"
)
public class Customer {
    @Id
    private Long id;

    @OneToMany(mappedBy = "customer")
    private Map<Long, Order> orders;

    // getters and setters
}

```

This way we can easily use this query in our code by calling `entityManager.createNamedQuery("Customer.findOrder")`.

From Clause - Fetch Join

let's say we have another entity called "Department" that is related to the "User" entity through a Many-to-One relationship. We can use a `FETCH JOIN` to retrieve both entities together in a single query:

```
SELECT u FROM User u JOIN FETCH u.department
```

In this query, we're using a `JOIN` clause to join the "User" entity with its related "Department" entity, and we're using the `FETCH` keyword to indicate that we want to retrieve the "Department" entity eagerly (i.e., load it in memory along with the "User" entity).

To use this JPQL query in a JPA application, we can define a named query using the `@NamedQuery` annotation on our entity class:

```
@Entity
@NamedQuery(
    name = "User.findAllWithDepartments",
    query = "SELECT u FROM User u JOIN FETCH u.department"
)
public class User {
    // ...
}
```

In this code snippet, we're defining a named query called `User.findAllWithDepartments` that corresponds to the JPQL query we wrote earlier. We can then use the `EntityManager` to execute this named query and retrieve the results:

```
TypedQuery<User> query = em.createNamedQuery("User.findAllWithDepartments",
    User.class);
List<User> users = query.getResultList();
```

In this code snippet, we're creating a `TypedQuery` object using the named query we defined earlier, and we're specifying that the result type should be a list of `User` entities. We can then call the `getResultList()` method to execute the query and retrieve the results.

Where Clause

The `WHERE` clause is used to specify a condition that must be met for each record returned by the query.

For example, imagine you have a `Person` entity that has a `name` attribute. You could use JPQL and the `WHERE` clause to find all `Person` entities with the name "John":

```
SELECT p FROM Person p WHERE p.name = 'John'
```

Next, let's talk about passing parameters to the entity manager. There are two ways to pass parameters: positional parameters and named parameters. A positional parameter is represented by a question mark (?) in the query, and values are passed in the order they appear in the query. A named parameter is represented by a colon (:) followed by a name, and values are passed using the `setParameter()` method.

Here's an example of a JPQL query using a named parameter:

```
@NamedQuery(  
    name="findPersonByName",  
    query="SELECT p FROM Person p WHERE p.name = :name"  
)
```

In this example, the named parameter `name` is used in the `WHERE` clause to find a `Person` with a specific name. To pass a value to this query using the entity manager, you would use the `setParameter()` method like so:

```
String name = "John";  
TypedQuery<Person> query = em.createNamedQuery("findPersonByName", Person.class);  
query.setParameter("name", name);  
List<Person> people = query.getResultList();
```

In this code, we're setting the `name` parameter to "John" and then executing the query using the entity manager. The results are returned as a list of `Person` entities.

Where Clause - Between Operator

The "`between operator`" is used in a where clause to specify a range of values that data should fall within. For example, if we have a database of products with a "`price`" field, we could use the between operator to retrieve all products with prices between \$10 and \$20.

Here's an example of how you can use `@NamedQuery` and entity manager to write a JPQL query with a where clause and between operator:

Let's say we have an entity called "`Product`" with fields `id`, "`name`", and "`price`". We want to retrieve all products with prices between \$10 and \$20. Here's how we can do it:

1. First, we define our named query using the `@NamedQuery` annotation in the Product entity class:

```
@Entity  
@NamedQuery(name = "Product.findInRange", query = "SELECT p FROM Product p WHERE  
p.price BETWEEN :minPrice AND :maxPrice")  
public class Product {  
    // entity fields and methods go here  
}
```

In this named query, we're selecting all instances of the Product class (denoted by "`p`") where the price field is between two parameters `:minPrice` and `:maxPrice`.

1. Next, we can use the EntityManager class to execute this named query in our code:


```

EntityManager em = // get entity manager instance
TypedQuery<Product> query = em.createNamedQuery("Product.findInRange",
Product.class);
query.setParameter("minPrice", 10);
query.setParameter("maxPrice", 20);
List<Product> results = query.getResultList();

```

In this code, we're creating a TypedQuery object that will execute our named query "Product.findInRange" and return a list of Product objects. We then set the values of the ":minPrice" and ":maxPrice" parameters to 10 and 20, respectively. Finally, we call getResultList() to retrieve all products that match our query.

Where Clause - Like Operator

In JPQL, the "like operator" is used in a where clause to match patterns in string values. This operator is useful when you want to retrieve data that matches specific text patterns.

Here's an example of how you can use @NamedQuery and entity manager to write a JPQL query with a where clause and like operator:

Let's say we have an entity called "Customer" with fields "id", "firstName", and "lastName". We want to retrieve all customers whose last name starts with "Smi". Here's how we can do it:

1. First, we define our named query using the @NamedQuery annotation in the Customer entity class:

```

@Entity
@NamedQuery(name = "Customer.findByLastNamePattern", query = "SELECT c FROM Customer
c WHERE c.lastName LIKE :pattern")
public class Customer {
    // entity fields and methods go here
}

```

In this named query, we're selecting all instances of the Customer class (denoted by "c") where the lastName field matches a pattern specified by the :pattern parameter.

1. Next, we can use the EntityManager class to execute this named query in our code:

```

EntityManager em = // get entity manager instance
TypedQuery<Customer> query = em.createNamedQuery("Customer.findByLastNamePattern",
Customer.class);
query.setParameter("pattern", "Smi%");
List<Customer> results = query.getResultList();

```

In this code, we're creating a TypedQuery object that will execute our named query "Customer.findByLastNamePattern" and return a list of Customer objects. We then set the value of the :pattern parameter to Smi%, which will match any last name that starts with "Smi". Finally, we call getResultList() to retrieve all customers that match our query.

Done

☒ Day 28/90 ==> [JavaEE: Day 28/90 - JPQL \(Java Persistence Query Language\)](#).