DMIT2015

Jakarta Data Repositories with Hibernate

Learning Objectives

- Upon completion of this lesson, you will be able to:
  - Understand the purpose and benefits of Jakarta Data.
  - Configure a Jakarta EE project to use Jakarta Data.
  - Utilize IntelliJ IDEA to "Create JPA entities from DB".
  - Write basic queries using automatic query methods (`@Find`).
  - Write queries using annotated query methods (`@Query,@HQL,@SQL`)
  - Filter, sort, limit, and join data using Jakarta Data queries.

Required Materials

- IntelliJ IDEA Ultimate Edition
- Assignment 4 project setup instructions
- Oracle 23ai database with Human Resources schema and Customer Orders schema

DEMO ASSIGNMENT 4

Why Jakarta Data?

- Imagine you're building a new application and need to interact with a database. You're familiar with JPA, but you're hearing about Jakarta Data. Why would you consider it? What problems might it solve?

Jakarta Data

- Jakarta Data works with entities defined by Jakarta Persistence for relational databases and entities defined by Jakarta NoSQL for NoSQL databases.
  - An entity is a Java class which represents data in a relational database table.
  - An entity has attributes (properties or fields) which map to columns of the table.
- Jakarta Data defines a simple subset of Jakarta Persistence Query Language (JPQL) called Jakarta Data Query Language (JDQL).
- Persistence contexts are stateless.
- The gateway is the user-written `@Repository` interface.
- Persistence operations with user-written methods annotated with `@Find, @Insert, @Update, @Save, @Delete`.
- Validation of JDQL is at compile time.

Persistence vs Data

https://docs.jboss.org/hibernate/orm/6.6/repositories/html_single/Hibernate_Data_Repositories.html

	Jakarta Persistence	Jakarta Data
Persistence Context	Stateful	Stateless
Gateway	`EntityManager` interface	User-written `@Repository` interface
Underlying implementation	`Session`	`StatelessSession`
Persistence Operations	Generic methods like `persist()`, `find()`, `merge()`, `remove()`	Type safe user-written methods annotated `@Insert`, `@Update`, `@Save`, `@Delete`
Updates	Usually implicit	Always explicit by calling `@Update` method
Lazy fetching	Implicit	Explicit using `@StatelessSession.fetch()`
Validation of JPQL	Runtime	Compile time

Maven Dependencies

- Maven dependencies for Jakarta Data:

```
<dependency>
    <groupId>jakarta.data</groupId>
    <artifactId>jakarta.data-api</artifactId>
    <version>1.0.1</version>
</dependency>
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.6.33.Final</version>
</dependency>
```

Annotation Processor

- Jakarta Data requires the annotation processor to run when your project is compiled.
- Maven plugin for annotation processor:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.14.0</version>
    <configuration>
        <annotationProcessorPaths>
            <path>
                <groupId>org.hibernate.orm</groupId>
                <artifactId>hibernate-jpamodelgen</artifactId>
                <version>6.6.33.Final</version>
            </path>
            <!-- other annotation processors -->
        </annotationProcessorPaths>
    </configuration>
</plugin>
```

jboss-deployment-structure.xml

- Wildfly application server includes its own version of Hibernate ORM.
  - WildFly 36.0.1 includes Hibernate ORM 6.6.7.Final
- To use your own version of Hibernate ORM, create a file named **jboss-deployment-structure.xml** in "**src/main/webapp/WEB-INF**" directory of your Jakarta EE project to exclude the default Hibernate module.

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.2">
    <deployment>
        <exclusions>
            <!-- Exclude the default Hibernate module -->
            <module name="org.hibernate" />
        </exclusions>
    </deployment>
</jboss-deployment-structure>
```

Assignment 4 Time (15 min)

- Complete Project Setup

Oracle Database 23ai Free docker/podman container

- Demo will be using Human Resources schema from Oracle Database 23ai Free.
- You can start the Oracle container with the command:

  `sudo podman start dmit2015-oracle-free-lite`
- To fetch the logs from the Oracle container:

  `sudo podman logs dmit2015-oracle-free-lite`
- You can stop the Oracle container with the command:

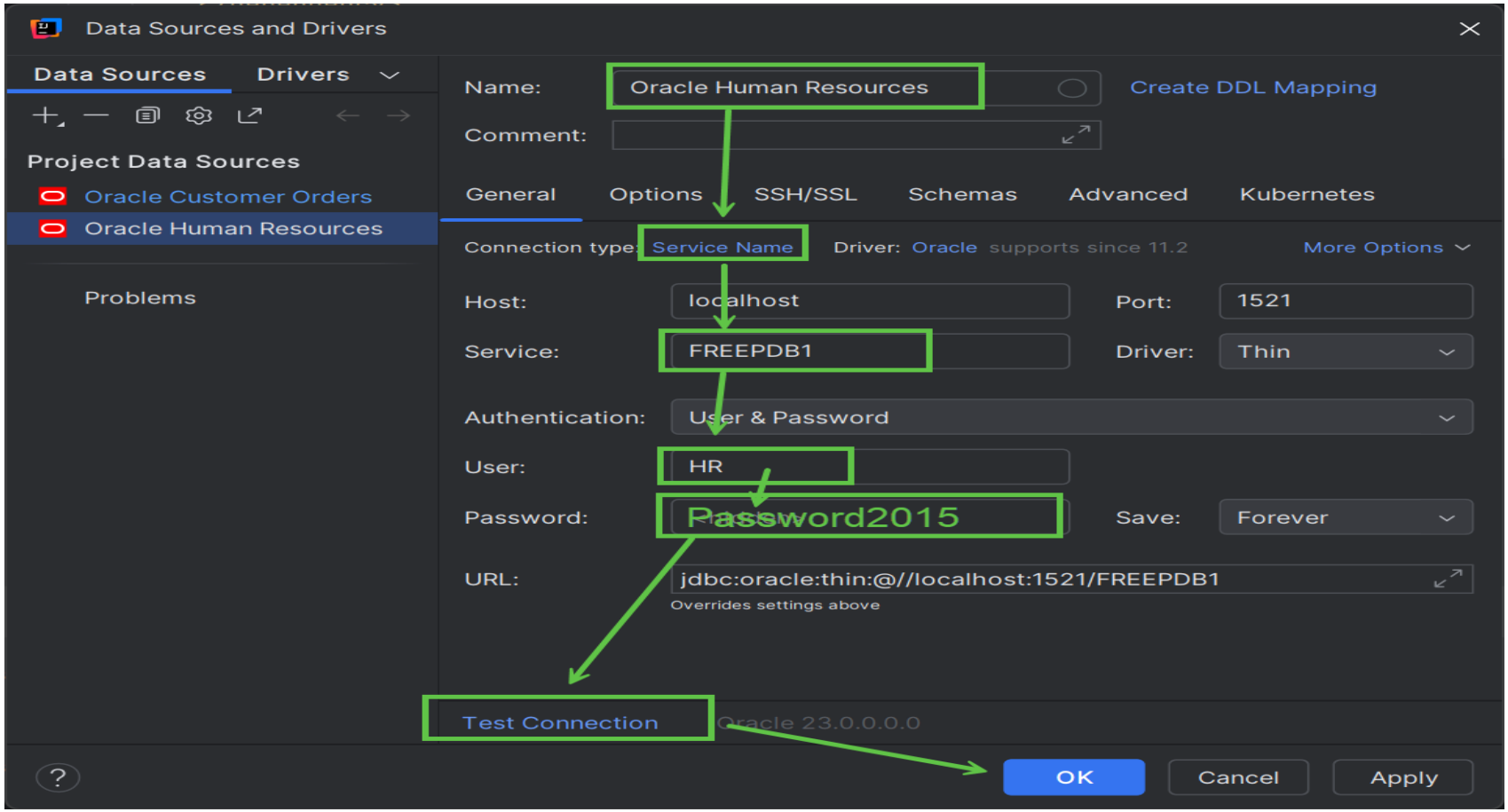
  `sudo podman stop dmit2015-oracle-free-lite`

Oracle Database 23ai Connection Settings

- Human Resources schema
  - Connection type: **Service Name**
  - Host: **`localhost`**
  - Service: **`FREEPDB1`**
  - User: **`HR`**
  - Password: **`Password2015`**
  - URL: **`jdbc:oracle:thin:@//localhost:1521/FREEPDB1`**
- Customer Orders schema
  - Connection type: **Service Name**
  - Host: **`localhost`**
  - Service: **`FREEPDB1`**
  - User: **`CO`**
  - Password: **`Password2015`**
  - URL: **`jdbc:oracle:thin:@//localhost:1521/FREEPDB1`**

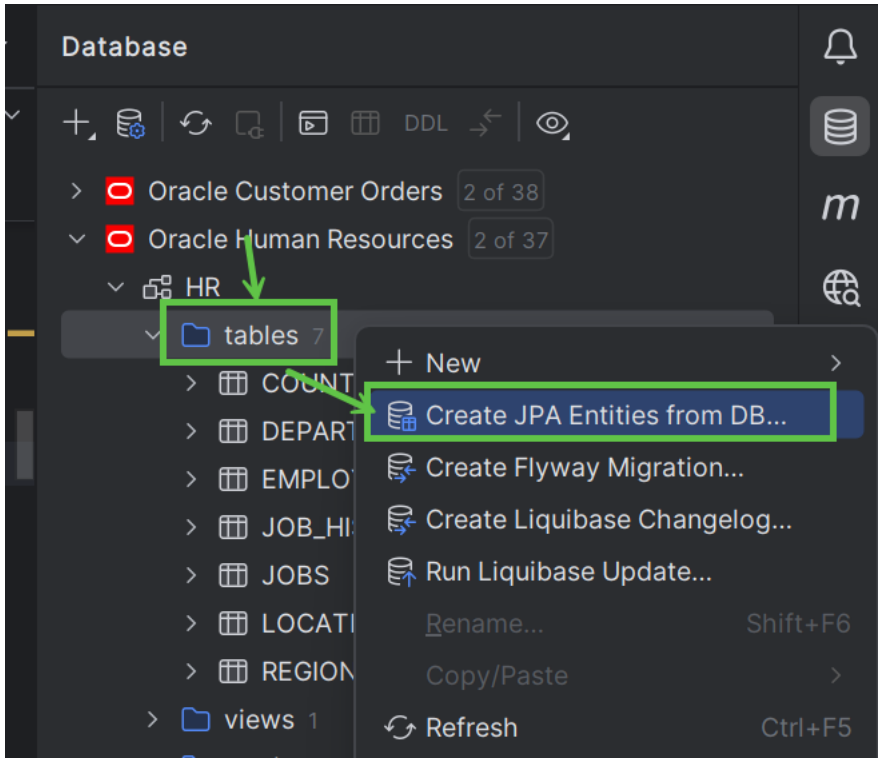
IntelliJ Database Window

- The **Database window** in IntelliJ IDEA can be used to browse the contents of a data source and to "Create JPA Entities from DB".
- The example below is a data source to access the Oracle HR database schema



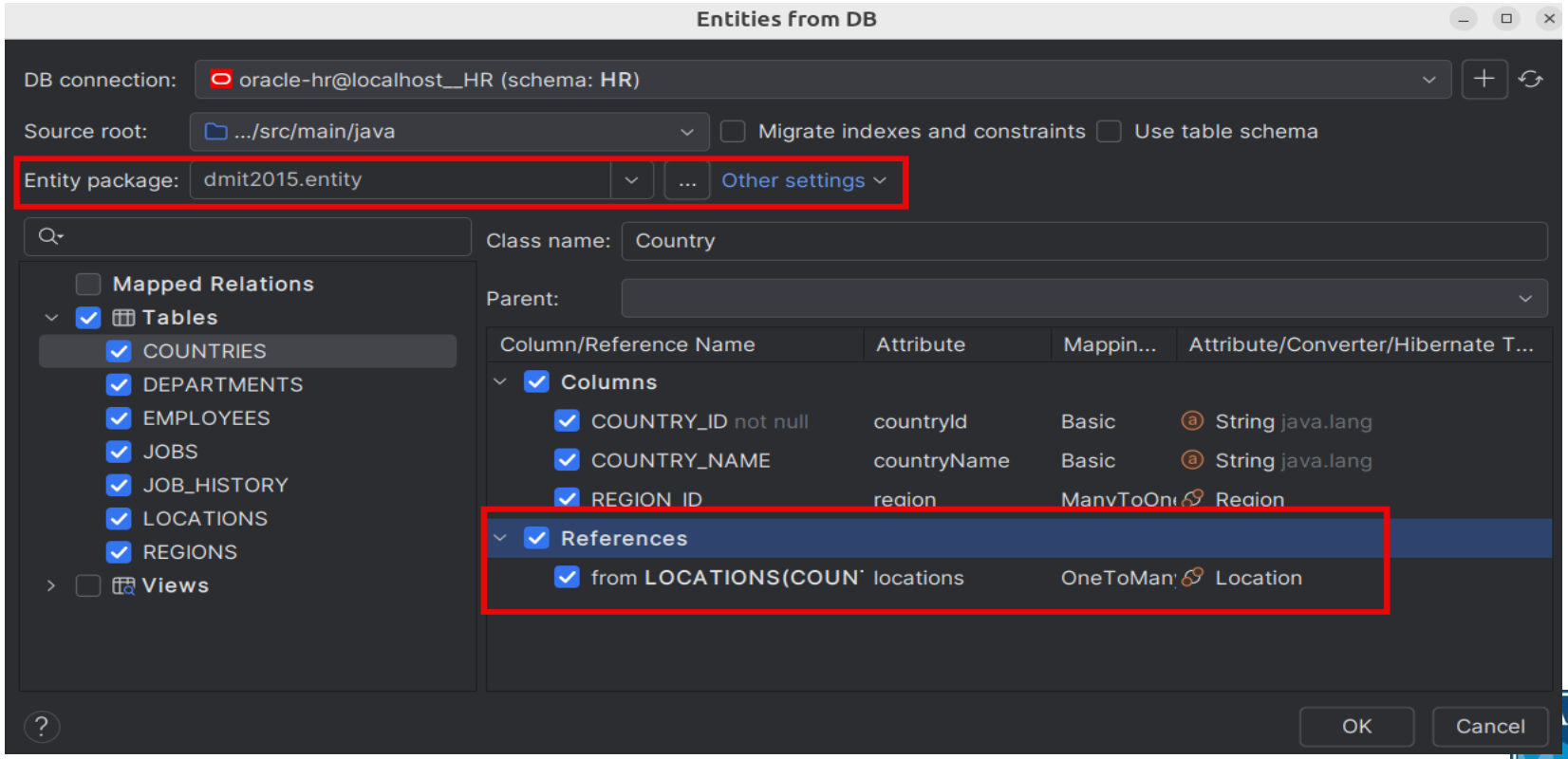
Create JPA Entities from DB (1)

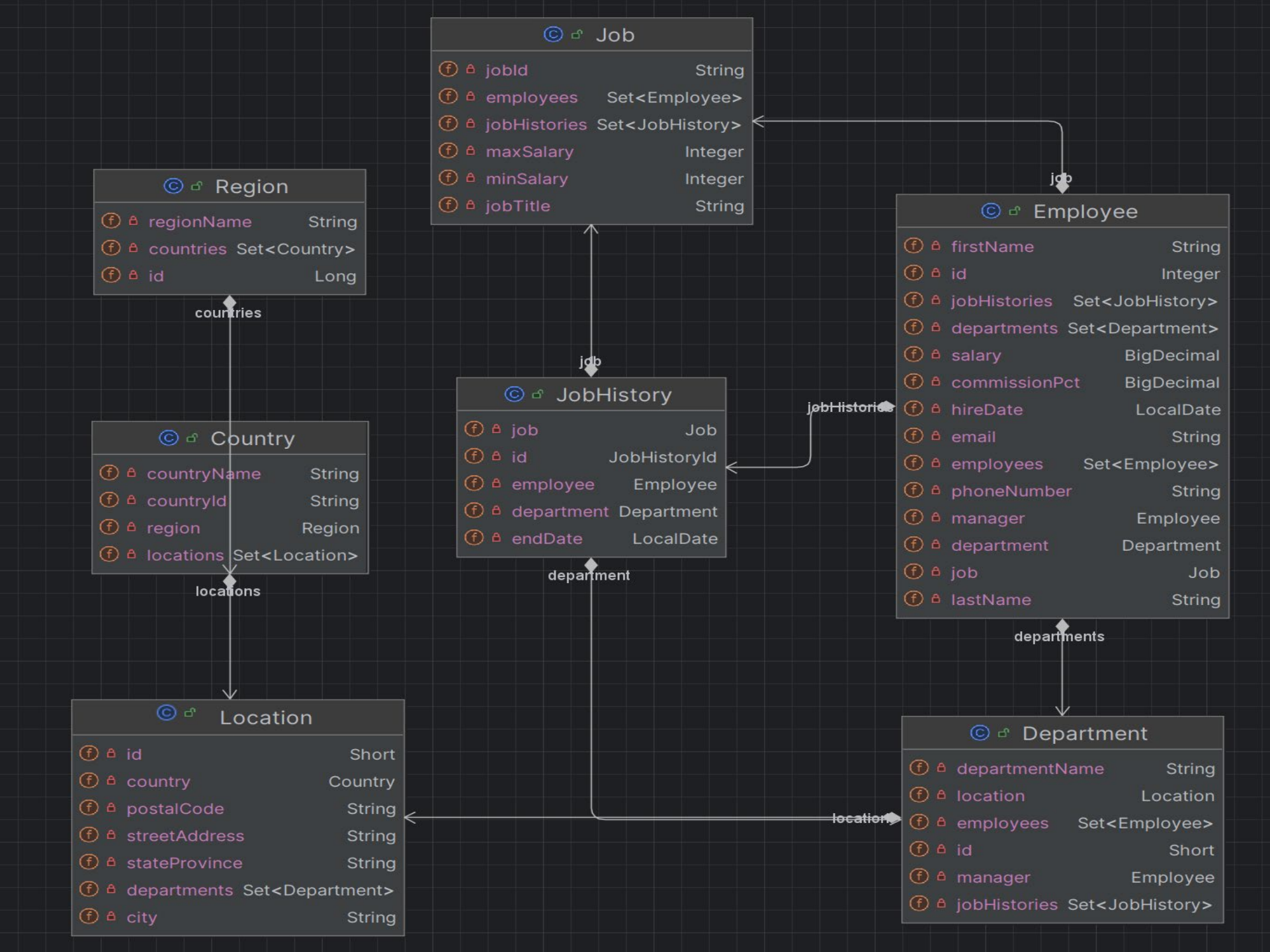
- The **Database window** in IntelliJ IDEA can be used to "Create JPA Entities from DB".
- Navigate to the tables folder of the data source, right-mouse click on tables folder then select "Create JPA Entities from DB"



Create JPA Entities from DB (2)

- Change the "**Entity package**" to the location to where you want to store the generated classes
- Select "**Tables**"
- Select each table name then check the "**References**" option




Job
jobId String
employees Set<Employee>
jobHistories Set<JobHistory>
maxSalary Integer
minSalary Integer
jobTitle String
Region
regionName String
countries Set<Country>
id Long
countries
job
Employee
firstName String
id Integer
jobHistories Set<JobHistory>
departments Set<Department>
salary BigDecimal
commissionPct BigDecimal
hireDate LocalDate
email String
employees Set<Employee>
phoneNumber String
manager Employee
department Department
job Job
lastName String
JobHistory
job Job
id JobHistoryId
employee Employee
department Department
endDate LocalDate
jobHistories
Country
countryName String
countryId String
region Region
locations Set<Location>
locations
department
departments
Location
id Short
country Country
postalCode String
streetAddress String
stateProvince String
departments Set<Department>
city String
location
Department
departmentName String
location Location
employees Set<Employee>
id Short
manager Employee
jobHistories Set<JobHistory>

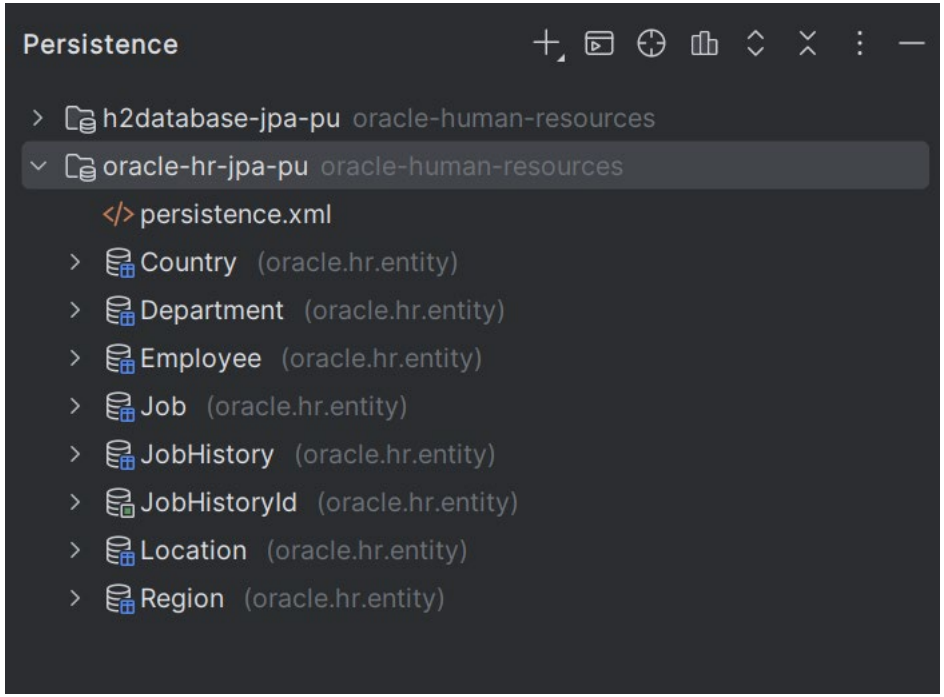
persistence.xml update

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence https://jakarta.ee/xml/ns/
             version="3.0">

    <persistence-unit name="oracle-jpa-hr-pu" transaction-type="JTA">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <jta-data-source>java:app/datasources/OracleHrDS</jta-data-source>

        <class>dmit2015.entity.Department</class>
        <class>dmit2015.entity.Location</class>
        <class>dmit2015.entity.Employee</class>
        <class>dmit2015.entity.Country</class>
        <class>dmit2015.entity.Job</class>
        <class>dmit2015.entity.JobHistory</class>
        <class>dmit2015.entity.Region</class>
        <class>dmit2015.entity.JobHistoryId</class>

        <properties>
            <property name="hibernate.type.preferred_instant_jdbc_type" value="TIMESTAMP"/>
        </properties>
    </persistence-unit>


</persistence>
```

You need to manually specify which classes are entities.

IntelliJ Persistence Console

- The **Persistence window** in IntelliJ IDEA includes a JPA Console that you can use to write and execute JPQL statements
- To open JPA Console, right-mouse click on the persistence unit then select **JPA Console (Ctrl+Shift+F10)**
  - If there is no context option for "JPA Console" then right-mouse click on persistence unit then select "Edit Persistence Unit" and assign a "DB Connection"



Assignment 4 Time (15 min)

- Complete Development Requirement 1 Entity Generation

JPQL/JDQL vs SQL (1)

- Keywords in both SQL and Jakarta Persistence Query Language (JPQL) are not case-sensitive.
- JPQL is **case-sensitive** and access Java **class** names and **attribute** names of the entity class. You **must** use the entity alias (e.g., e) to reference entity attributes. No direct table names.

SQL: `select * from Employee`

JPQL: `select e from Employee e`

- Filtering (WHERE clause) – **must** use the entity alias (e.g., e) to reference entity attributes.

SQL: `select first_name, last_name, salary from Employee where salary > 10000;`

JPQL: `select e.firstName, e.lastName, e.salary from Employee e where e.salary > 10000;`

JPQL vs SQL (2)

- Sorting (ORDER BY clause) – identical

SQL: `select first_name, last_name, salary from Employee order by salary desc, last_name, first_name;`

JPQL: `select e.firstName, e.lastName, e.salary from Employee e order by e.salary desc, e.lastName, e.firstName;`

- Limiting (TOP, ROWNUM or FETCH FIRST)

T-SQL: `select top 3 first_name, last_name, salary from Employee order by salary desc;`

PL-SQL 12c+: `select first_name, last_name, salary from Employee order by salary desc fetch first 3 rows only;`

JPQL: `select e.firstName, e.lastName, e.salary from Employee e order by e.salary desc, e.lastName, e.firstName fetch first 3 rows only;`

JPQL vs SQL (3)

- Aggregate Functions (GROUP BY, HAVING)

SQL:
```
SELECT department_id, COUNT(*) AS employee_count FROM employees GROUP BY department_id HAVING COUNT(*) > 10;
```

JPQL:
```
SELECT d.id, COUNT(e) AS employeeCount FROM Employee e join e.department d GROUP BY d.id HAVING COUNT(e) > 10
```

  - The JOIN syntax is different
  - You need to use the entity alias when counting entities.

- Joining Data

SQL:
```
SELECT e.first_name, e.last_name, d.department_name FROM employees e INNER JOIN departments d ON e.department_id = d.department_id;
```

JPQL:
```
SELECT e.firstName, e.lastName, d.name FROM Employee e JOIN e.department d
```

  - JPQL uses Implicit Joins
  - JOIN lazy loading the related entity data is only loaded when you explicitly access a property of that related entity in your application code.
  - No ON Clause.

JPQL:
```
SELECT e FROM Employee e JOIN FETCH e.department
```

  - JOIN FETCH eagerly loads the related entity data in a single query.

JDQL Examples

- The following examples uses the Oracle HR Schema
- You can use the **OBJECT** function to convert the query result to a Java object:

```
select OBJECT(j) from Job j
```

- The **OBJECT** keyword is optional, the same JPQL could be written as:

```
select j from Job j
```

- You can use the "order by" clause to sort the query results ascending:

```
select j from Job j order by j.jobTitle
```

- You can use the desc keyword to sort the query results descending:

```
select j from Job j order by j.jobTitle desc
```

- You can select specific properties of an entity object:

```
select j.jobTitle from Job j order by j.jobTitle
```

- You can filter results by an associated object property:

```
select e from Employee e where e.job.jobTitle = 'Programmer'
```

- You can use common aggregate functions:

```
select avg(e.salary) from Employee e where e.department.departmentName =
'Shipping'
```

JPQL/JDQL Aggregate Functions

Function	Description	Return Type
`COUNT`	Total number of records	`Long`
`MAX`	Records with the largest numeric value	Same as field to which applied
`MIN`	Records with the lowest numeric value	Same as field to which applied
`AVG`	Average of all numeric values in column	`Double`
`SUM`	Sum of all values in column	`Long` when applied to **integral types** `Double` when applied to **floating-point** `BigDecimal` when applied to **`BigDecimal`**
`COALESCE`	Create an expression that returns null if all its arguments evaluate to null, and the value of the first non-null argument otherwise.	

Repository interfaces

- A repository interface is an interface written by you and annotated with `@Repository`.
  - You can set `dataStore` attribute with the name of a persistence unit to use with JPA.
- The methods of a repository interface must fall into one the following categories:
  - default methods
  - lifecycle methods annotated `@Insert`, `@Update`, `@Delete`, or `@Save`
  - automatic query methods annotated `@Find`
  - annotated query methods annotated `@Query` or `@SQL`
  - resource accessor methods

Repository interfaces (old way)

- You can create a repository interface for each entity.

```
@Repository(dataStore = "oracle-hr-jpa-pu")
public interface EmployeeRepository extends CrudRepository<Employee,Integer> {
  // query methods
}
```

Repository interfaces (new way)

- You can create a single interface for all entities or interface for related entities.

```
@Repository(dataStore = "oracle-hr-jpa-pu")
public interface EmployeeRepository extends Serializable {
  @Insert Employee addEmployee(Employee newEmployee);
  @Update Employee updateEmployee(Employee existingEmployee);
  @Delete void deleteEmployee(Employee existingEmployee);

  @Find List<Employee> employees();
  @Find Employee employee(Integer id);
  // The _ character in a parameter name is used to navigate associations
  @Find List<Employee> employeesByDepartment(String department_departmentName);
  @Find List<Employee> employeesByJob(String job_jobTitle);

 @Query("select e.firstName, e.lastName from Employee e where
e.department.departmentName = :deptName")
  List<EmployeeName> employeeNamesByDepartment(String deptName);
}
public record EmployeeName(String firstName, String lastName) {}
```

Injecting a repository

- You can use the `@Inject` annotation to inject a repository implementation in a CDI managed class.

```
// @ApplicationScoped/@SessionScoped/@ViewScoped/@RequestScoped
// @Named
public class SomeCdiManagedClass {

  @Inject EmployeeRepository _employeeRepository;

}
```

Lifecycle methods

- Jakarta Data defines four built-in lifecycle annotations
  - `@Insert` maps to `insert()`,
  - `@Update` maps to `update()`,
  - `@Delete` maps to `delete()`, and
  - `@Save` maps to `upsert()`.
- A lifecycle method usually accepts an instance of an entity type, and is usually declare `void`

```
@Insert
void add(Region newRegion);

@Update
void update(Region existingRegion);

@Delete
void delete(Region existingRegion);
```

Automatic Query Methods

- An automatic query method is usually annotated `@Find`.
- Retrieve an entity instance by its unique identifier.

```
@Find Employee employee(Integer id); // EmptyResultException thrown if no result
```

- Retrieve an `Optional` entity instance by its unique identifier.

```
@Find Optional<Employee> employee(Long id);
```

- Retrieve a list of Employee entity that match exactly `firstName`.

```
@Find @OrderBy("lastName")
List<Employee> employeesByLastName(String firstName);
```

- Retrieve a list of Employee entity that match using `like`.

```
@Find
List<Employee> employeesByName(@Pattern String firstName, @Pattern String lastName);
```

- The _ character in a parameter may be used to navigate associations:

```
@Find
List<Employee> employeesByJobTitle(String job_jobTitle);
@Find
List<Employee> employeesByDepartmentName(String department_departmentName);
```

Annotated Query Methods (1)

- An annotated query method is declared using `@Query`
- Retrieve a list of Employee entity that match partial `jobTitle` using named parameters.

```
@Query("select e from Employee e where e.job.jobTitle like :jobTitle")
List<Employee> employeesByJobTitle(String jobTitle);
```

- Retrieve a list of Employee entity that match partial `id` using ordinal parameters.

```
@Query("select e from Employee e where e.department.id like ?1")
List<Employee> employeesByDepartmentid(Short departmentId);
```

- List each job title along with the average salary of employees in that job.

```
@Query("select e.job.jobTitle, avg(e.salary) from Employee e group by e.job.jobTitle")
List<JobAverageSalary> jobAverageSalary();
// public record JobAverageSalary(String jobTitle, Double averageSalary) {}
```

Annotated Query Methods (2)

- You can use the `join` keyword fetch a collection entity association. Write a query to display the department name and the number of employees in each department.

```
@Query("select d.departmentName, count(e.id) from Department d join
d.employees e group by d.departmentName order by d.departmentName")
List<DepartmentEmployeeCount> departmentEmployeeCount();
// public record DepartmentEmployeeCount(String departmentName, Long employeeCount) { }
```

- You can use the `join fetch` keyword fetch an entity association. Search for Employee by Department.id and fetch the entity associated attributes `department`, `job`, and `manager`.

```
@Query("select e from Employee e join fetch e.department join fetch e.job
join fetch e.manager where e.department.id = ?1")
List<Employee> employeesByDepartmentId(Short id);
```

Positional Parameters

- JDQL/JPQL positional parameters are marked with **?**
  - Set using position, which starts with index **1**

```
@Query("select e from Employee e where e.hireDate > ?1 and
e.job.jobTitle like ?2 ")
List<Employee> employeesByHireDateAndJobTitle(LocalDate hireDate,
@Pattern String jobTitle);
```

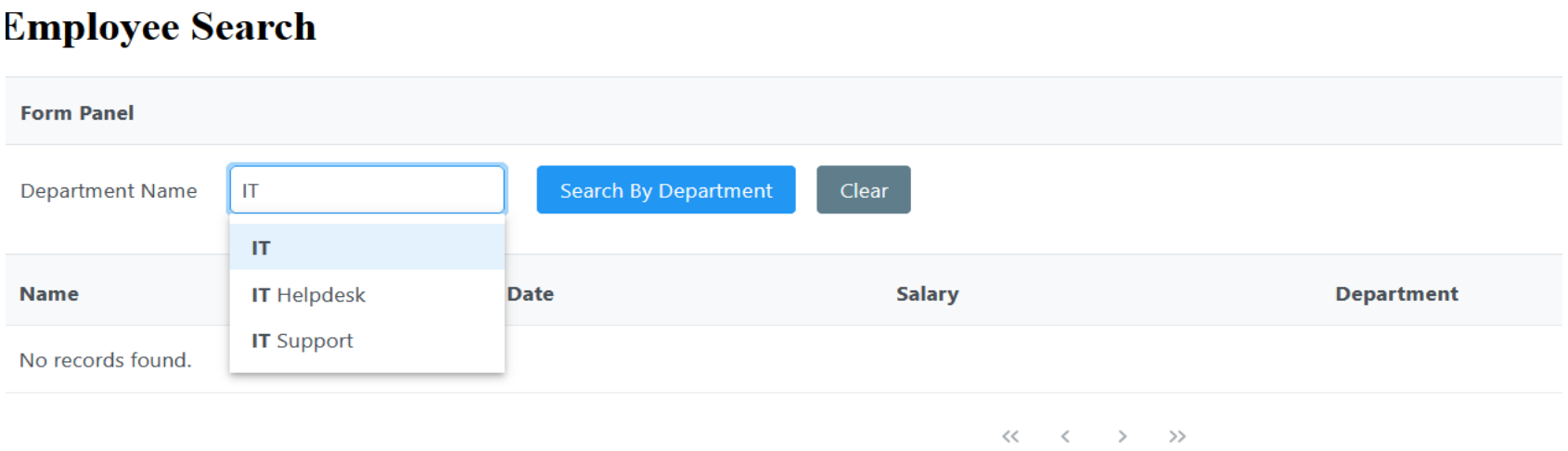
Named Parameters

- JDQL/JPQL named parameters uses symbolic name using colon and parameter name `:parametername`

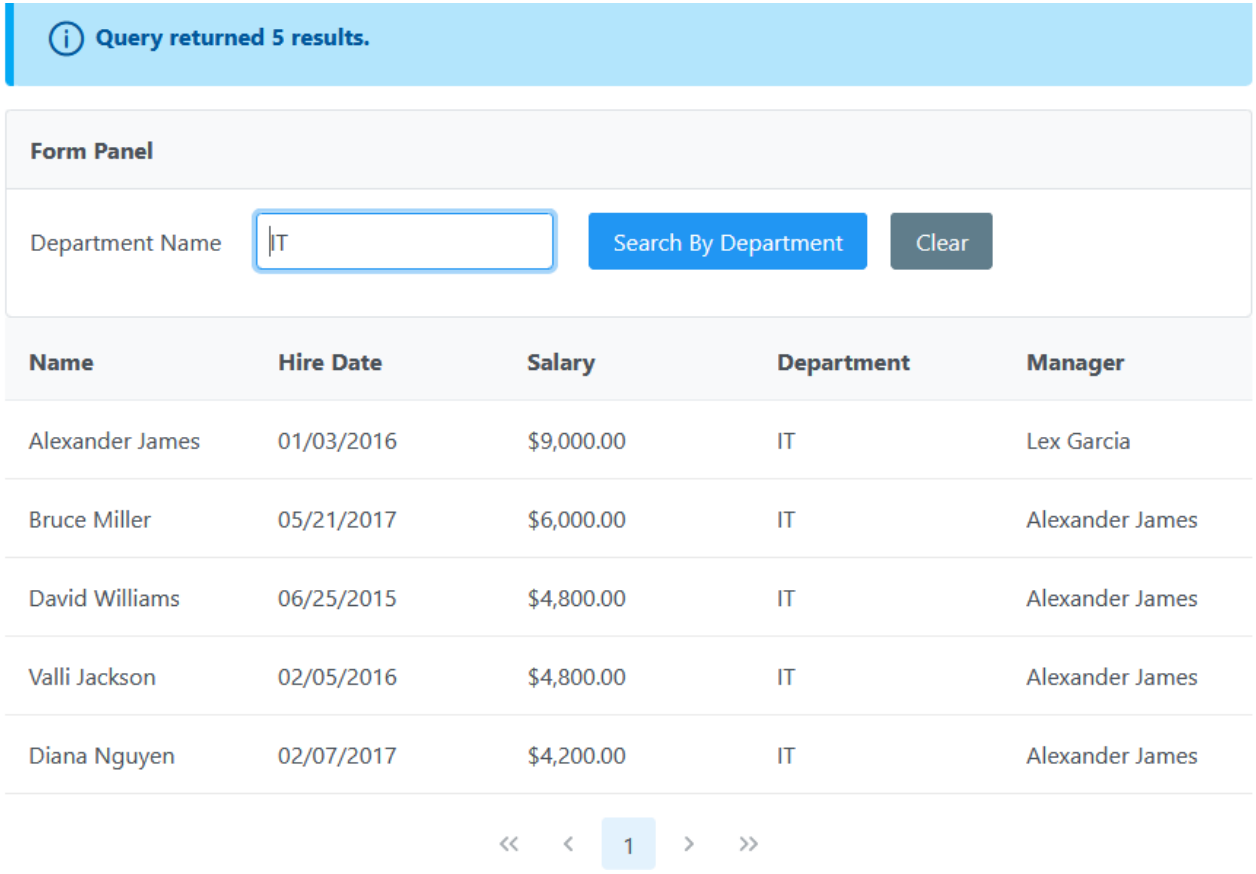
```
@Query("select e from Employee e where e.hireDate > :hireDate and
e.job.jobTitle like :jobTitle ")
List<Employee> employeesByHireDateAndJobTitle(LocalDate hireDate,
@Pattern String jobTitle);
```

Example

- Create a Jakarta Faces page to search for employees by department name using an PrimeFaces AutoComplete component.



Search Results



Step 1: EmployeeRepository.java

```
@Repository(dataStore = "oracle-hr-jpa-pu")
public interface EmployeeRepository extends Serializable {

  @Query("select d from Department d where lower(d.departmentName) like lower(?1) order by
d.departmentName")
  List<Department> departmentsBy(String departmentName);

  @Query("select e from Employee e join fetch e.department join fetch e.job join fetch e.manager where
e.department.id = ?1")
  List<Employee> employeesByDepartmentId(Short id);

  @Find
  Department departmentByDepartmentId(Short id);

}
```

Assignment 4 Time (10 min)

- Complete Development Requirement 2 Product Repository
- Complete Development Requirement 2 Inventory Repository
- Can be combined into a single repository (ie. CustomerOrdersRepository)

FacesConverter

- A `FacesConverter` is a specialized type of converter in Jakarta Faces that's designed to handle the conversion between a UI component's value (typically a String) and a backing bean's property (which could be any object type).
- A `FacesConverter` extends the `Converter` interface and must implement the following methods:
  - `getAsString(FacesContext context, UIComponent component, Object value)`: Converts the object `value` into a `String` representation suitable for display in the UI. This is called when the component's value needs to be rendered.
  - `getAsObject(FacesContext context, UIComponent component, String value)`: Converts the `String` representation from the UI back into an object. This is called when the user enters data into the component and the value needs to be processed by the backing bean.

Step 2: DepartmentConverter.java (1)

```
@Named
@ApplicationScoped
@FacesConverter(value = "departmentConverter", managed = true)
public class DepartmentConverter implements Converter<Department> {
    @Inject
    private EmployeeRepository _employeeRepository;

    @Override
    public Department getAsObject(FacesContext context, UIComponent component, String value) {
        if (value != null && !value.isBlank()) {
            try {
                Short departmentId = Short.parseShort(value);
                return _employeeRepository.departmentByDepartmentId(departmentId);
            } catch (NumberFormatException e) {
                throw new ConverterException(new
FacesMessage(FacesMessage.SEVERITY_ERROR,"Conversion Error","Not a valid department."));
            }
        } else {
            return null;
        }
    }
```

Step 2: DepartmentConverter.java (2)

```
    @Override
    public String getAsString(FacesContext context, UIComponent component, Department value) {
        if (value != null) {
            return value.getId().toString();
        } else {
            return null;
        }
    }
}
```

Assignment 4 Time (10 min)

- Create a ProductConverter (10 minutes)

Step 3: EmployeeQueryView.java (1)

```
@Named("currentEmployeeQueryView")
@ViewScoped
public class EmployeeQueryView implements Serializable {
  @Getter @Setter
  private Department selectedDepartment;

  @Getter
  private List<Employee> queryResults;

  @Inject
  private EmployeeRepository _employeeRepository;
```

Step 3: EmployeeQueryView.java (2)

```
public List<Department> completeDepartment(String query) {
  return _employeeRepository.departmentsBy("%" + query + "%");
}

public void onSearchByDepartment() {
  try {
    queryResults = _employeeRepository.employeesByDepartmentId(selectedDepartment.getId());
    Messages.addGlobalInfo("Query returned {0} results.", queryResults.size());
  } catch (Exception ex) {
    handleException(ex);
  }
}

public void onClear() {
  queryResults = null;
  selectedDepartment = null;
}
```

Step 3: EmployeeQueryView.java (3)

```
  /**
   * This method is used to handle exceptions and display root cause to user.
   *
   * @param ex The Exception to handle.
   */
  protected void handleException(Exception ex) {
    var details = new StringBuilder();
    Throwable causes = ex;
    while (causes.getCause() != null) {
      details.append(ex.getMessage());
      details.append("    Caused by:");
      details.append(causes.getCause().getMessage());
      causes = causes.getCause();
    }
    Messages.create(ex.getMessage()).detail(details.toString()).error().add("errors");
  }

}
```

Step 4: Employee.java

- Add a method to `Employee.java` to return the full name of the employee.

```
public String getFullName() {
    return String.format("%s %s", firstName, lastName);
}
```

Step 5: query.xhtml (1)

```
<h1>Employee Search</h1>
<p:messages id="messages">
    <p:autoUpdate/>
</p:messages>
<h:form prependId="false">
  <p:focus context="formPanel"/>
  <p:panel id="formPanel" header="Form Panel">
    <div class="ui-fluid">
      <div class="formgroup-inline">
        <div class="field grid">
          <p:outputLabel for="@next" styleClass="col-fixed" >Department Name</p:outputLabel>
          <div class="col">
            <p:autoComplete id="selectedDepartment"
               value="#{currentEmployeeQueryView.selectedDepartment}"
               completeMethod="#{currentEmployeeQueryView.completeDepartment}"
               var="department" itemLabel="#{department.departmentName}" itemValue="#{department}"
               converter="#{departmentConverter}"
               forceSelection="true"
               scrollHeight="250"
               />
          </div>
        </div>
```

Step 5: query.xhtml (2)

```
        <div class="field">
          <p:commandButton value="Search By Department" styleClass="col-fixed"
              action="#{currentEmployeeQueryView.onSearchByDepartment()}"
              update=":messages :growl @form dataTable"
           >
           </p:commandButton>
        </div>
        <div class="field">
            <p:commandButton value="Clear"
                          action="#{currentEmployeeQueryView.onClear()}"
                          update="@form :outputPanel dataTable"
                         styleClass="ui-button-secondary"
            >
            </p:commandButton>
          </div>
      </div>
    </div>
  </p:panel>
</h:form>
```

Step 5: query.xhtml (3)

```
<h:form prependId="false" id="outputPanel">
  <div class="card">
    <p:dataTable id="dataTable"
                 value="#{currentEmployeeQueryView.queryResults}"
                  var="currentItem"
                  reflow="true"
                  rowKey="#{currentItem.id}"
                   paginator="true" rows="10"
                   paginatorPosition="bottom">

      <p:column headerText="Name">
         <h:outputText value="#{currentItem.fullName}" />
      </p:column>

      <p:column headerText="Hire Date">
         <h:outputText value="#{currentItem.hireDate}">
            <f:convertDateTime type="localDate" pattern="MM/dd/yyyy"/>
         </h:outputText>
      </p:column>
```

Step 5: query.xhtml (4)

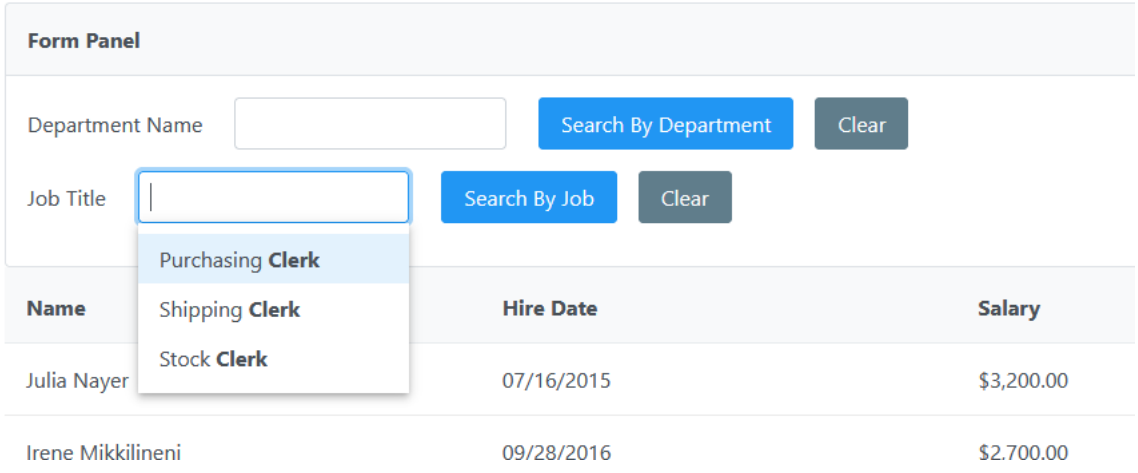
```
      <p:column headerText="Salary">
        <h:outputText value="#{currentItem.salary}">
          <f:convertNumber type="currency" locale="en_CA"/>
        </h:outputText>
      </p:column>

      <p:column headerText="Department">
         <h:outputText value="#{currentItem.department.departmentName}">
         </h:outputText>
         </p:column>

      <p:column headerText="Manager">
        <h:outputText value="#{(empty currentItem.manager) ? 'No Manager' :
currentItem.manager.fullName}" />
      </p:column>

    </p:dataTable>
  </div>
</h:form>
```

Assignment 4 Time (50 min)

- Complete Product Inventory Search Page
- Complete Navigation
- Complete Home Page
- Complete Deployment
- Complete Testing

Exercise

- Modify example to add the option to search for employees by job title using a PrimeFaces AutoComplete component.



Assignment 4 Time (30 min)

- Complete Map Integration

Resources

- Jakarta Data 1.0 Specification Document
- Jakarta Data 1.0 Specification Addendum
- Hibernate Data Repositories
- Getting started with Jakarta Data and Hibernate