CT5106

JPA (Java Persistence API)

Connecting to DB's

- There are number of ways we can connect to the DB – we will look at the first 2 for now:

1. Connecting IDE (NEtBeans) to DB (MySQL)
   - The purpose of this is to allow us to explore / query DB from with the IDE environment
2. Connecting application server (Payara) to DB (MySQL)
   1. The purpose of this is to allow our application to use JPA (which uses the JDBC driver) which relies on connection pools we create from within the Payara admin tool
3. Adding a dependency to Java (Maven) project to allow reverse engineering of database (to classes)

You need a MySQL database

- You can use the MySQL database you used for other modules
- If you don't have one, you can create it on the CS school intranet: https://www2.it.nuigalway.ie/intranet/
- You will need to use the admin userid and password that you receive when you set up the database

1. Connecting IDE (NetBeans) to DB (MySQL)

- We can create connections to databases from within NetBeans
- These connections can be used to run queries, see DB structure, insert / delete records etc
- These connections can also be used to engineer database tables and relations to create entity classes

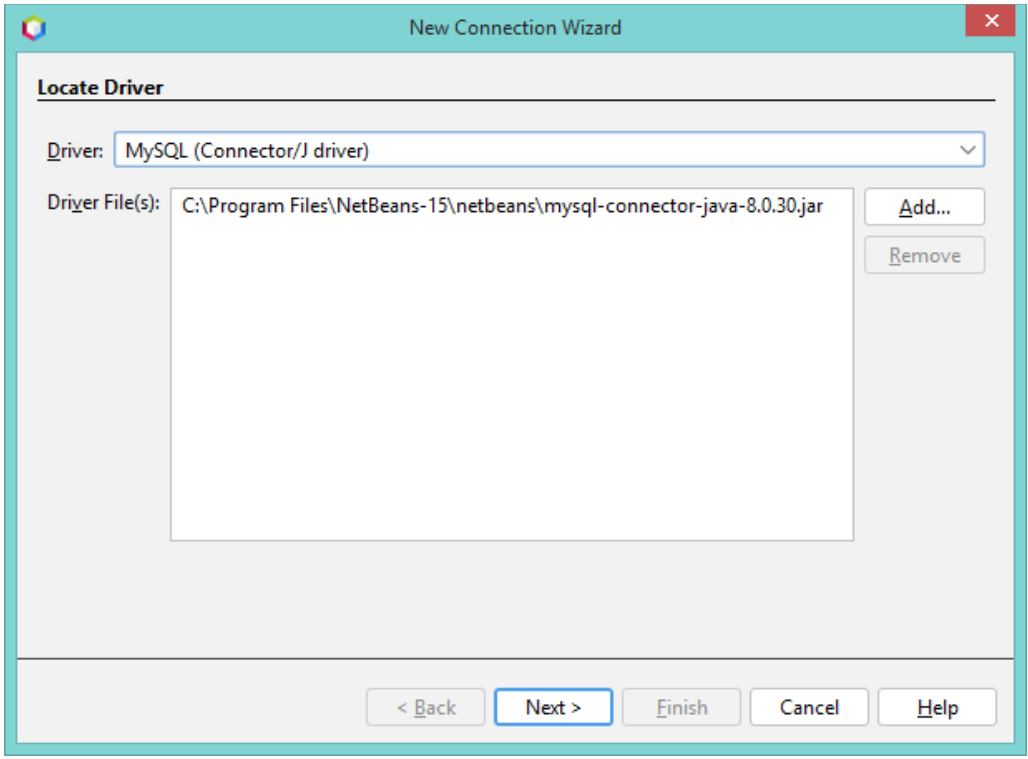
Download MySQL JDBC connector

- Go to https://dev.mysql.com/downloads/connector/j/
- Select 'Platform Independent' as the OS and then click on 'Go to Download Page'
- It will bring you to the MySQL Community Downloads page, and you should click on the 'Download Now' page,
  - but this will bring you to the Oracle site - you will need to log in to the Oracle site to access the download
- Download the 'mysql-connector-java-xxx.zip' file – I downloaded the **mysql-connector-java-8.0.30.zip** version
- Unzip and put somewhere you will remember – I put mine in the NetBeans installation folder

- Then go back to NetBeans

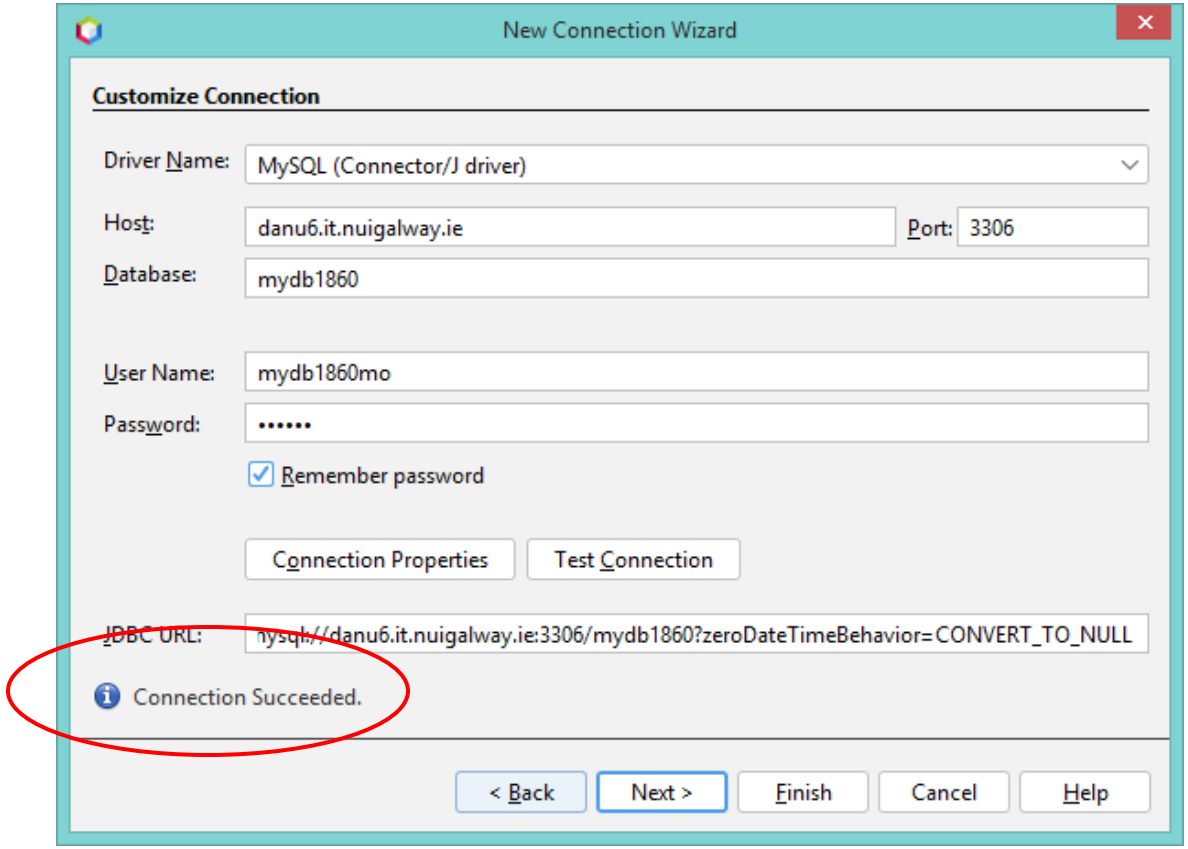
Start with connecting to MySQL

- In NetBeans
- Select Services tab
- Right-click on Databases
- Select 'New Connection'
- Select Driver: MySQL
- You will need to select the 'Add' button to add the driver file



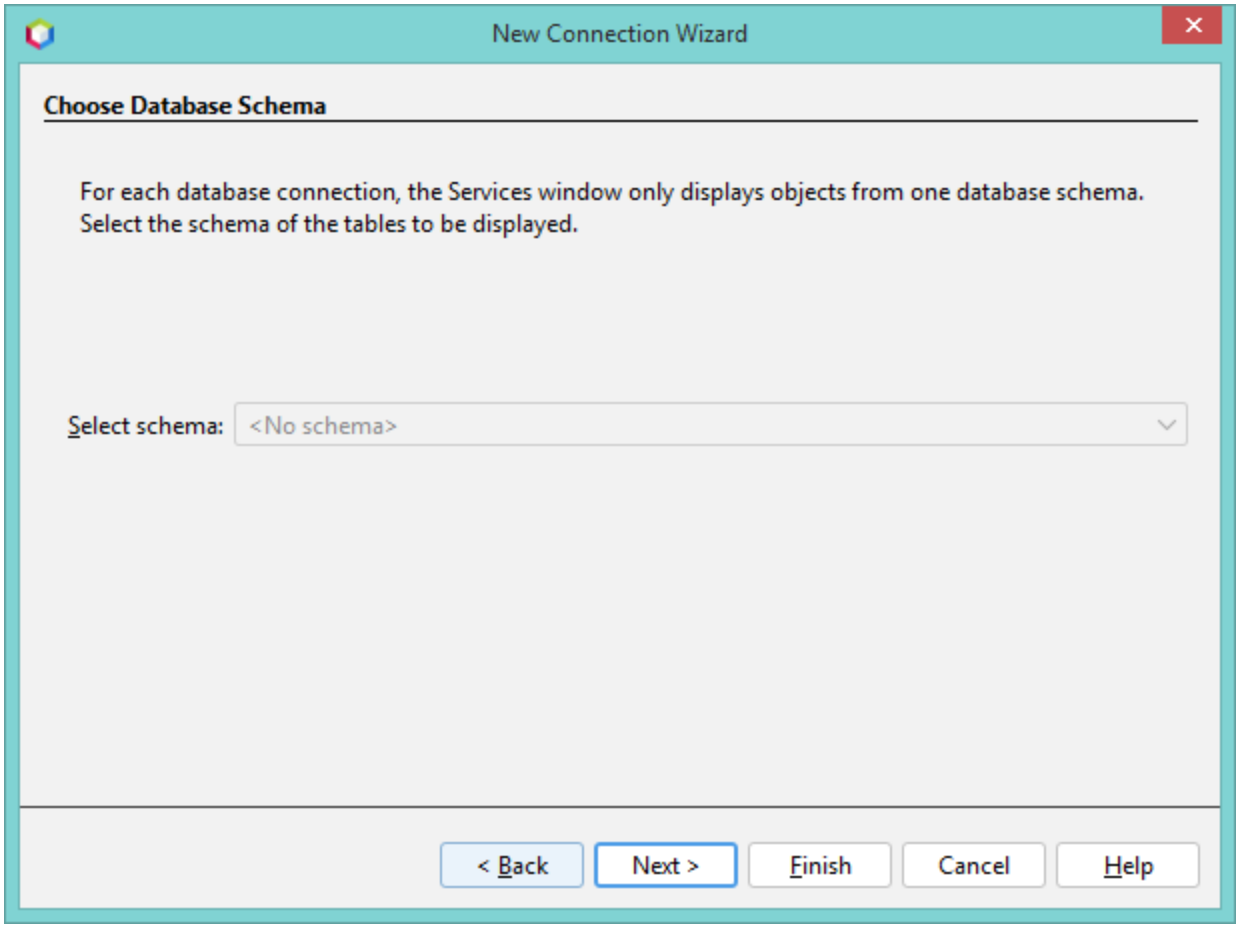
Set connection properties and test connection

- Enter your MySQL database connection properties
- Check the 'Remember password' box
- Click on 'Test Connection'
- Hopefully it will say 'Connection Succeeded'
- Click Next



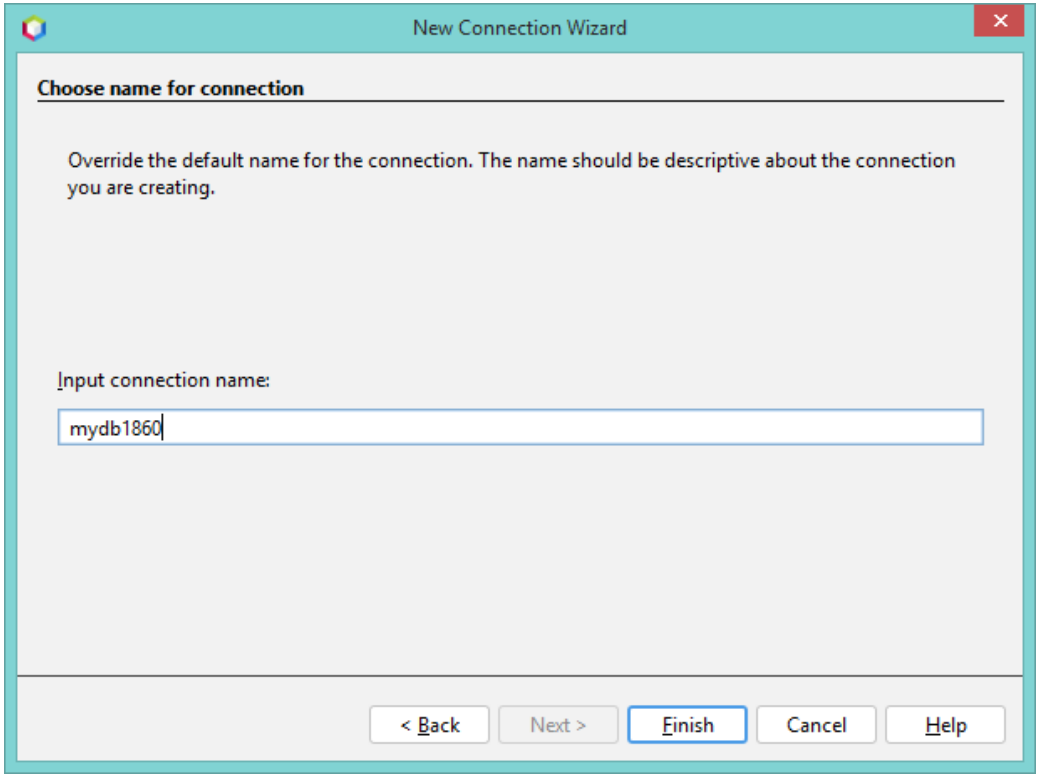
Choose database schema

- If you have only one schema in the database, this will show no schema to select, like this, so click on 'Next'



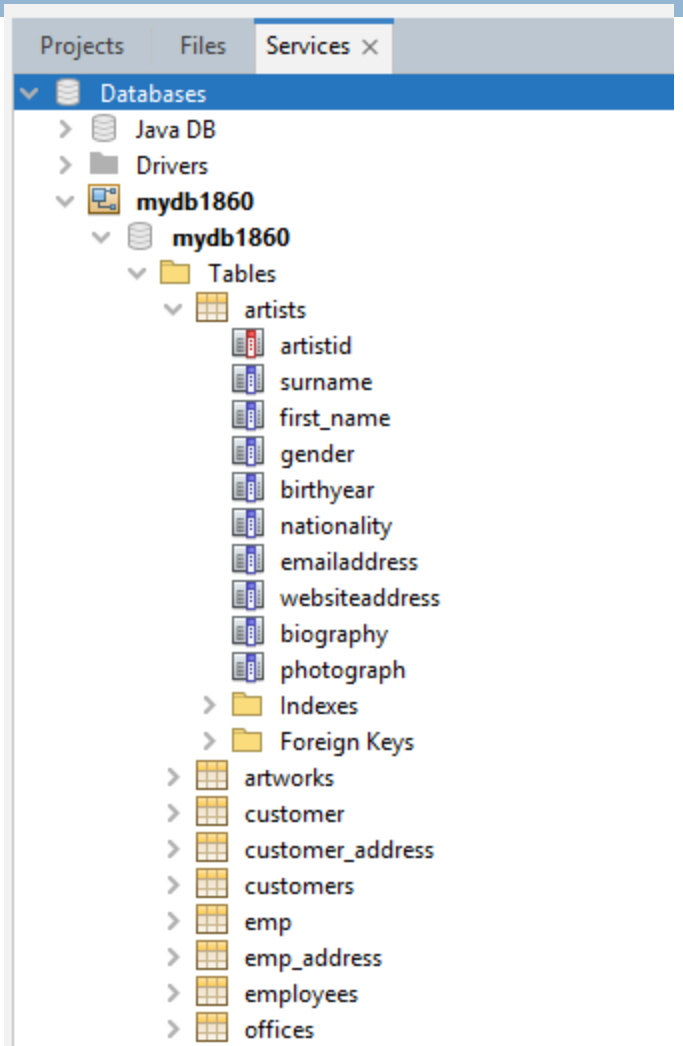
Give your connection a name

- Something short would be good!
- Then click ‘Finish’ and it’s done



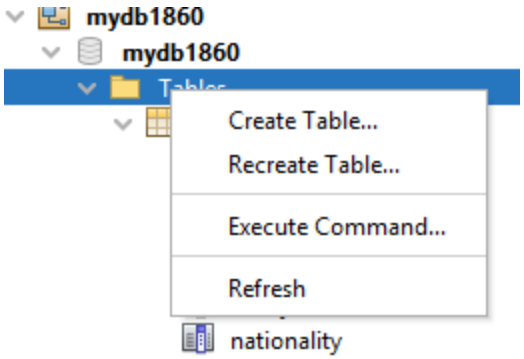
See what's in your database

- You should be able to drill down into the database and see the tables, views and procedures



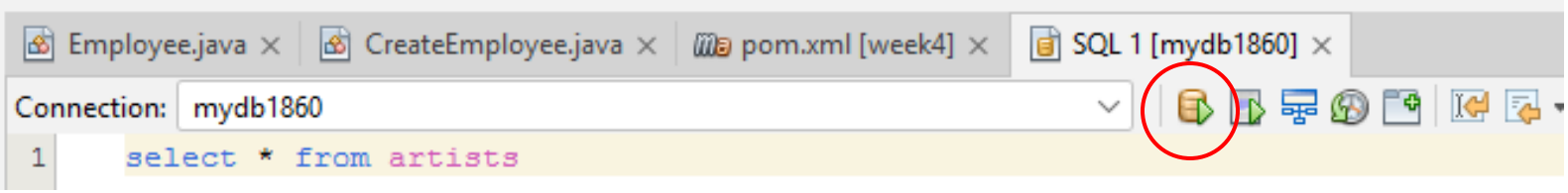
Run an SQL query

- Right click on 'Tables' and select 'Execute Command'



Write and run query

- This will open a query tab on the RHS
- Enter a simple query and click on the green triangle to run



Query results

- These will be shown in a tab on the bottom RHS

select * from artists

Max. rows: 100 Fetched Rows: 5 Matching Rows:

#	artistid	surname	first_name	gender	birthyear	
1	1003	Kahlo	Frida	F	1954	Mexican
2	1005	O'Keeffe	Georgia	F	1887	American
3	1010	Gentileschi	Artemisia	F	1593	Italian
4	3001	Frankenthaler	Helen	F	1928	American
5	4001	Cassatt	Mary	F	1844	American

2. Connecting application server (Payara) to DB (MySQL)

- We just added it to NetBeans so that we can see into your database from there, but to run applications that use JPA (and hence the jdbc connector to MySQL), we need to add the mysql connector .jar to the application server
- Go to your Payara server installation folder and open the /bin folder
- You should see just a few files there, including 'asadmin.bat'
- Double click on 'asadmin.bat' to run it

Add library in asadmin

- Enter the command ‘add-library’
- Then provide the location of the mysql connector jar file, like in the example below

C:\Windows\system32\cmd.exe

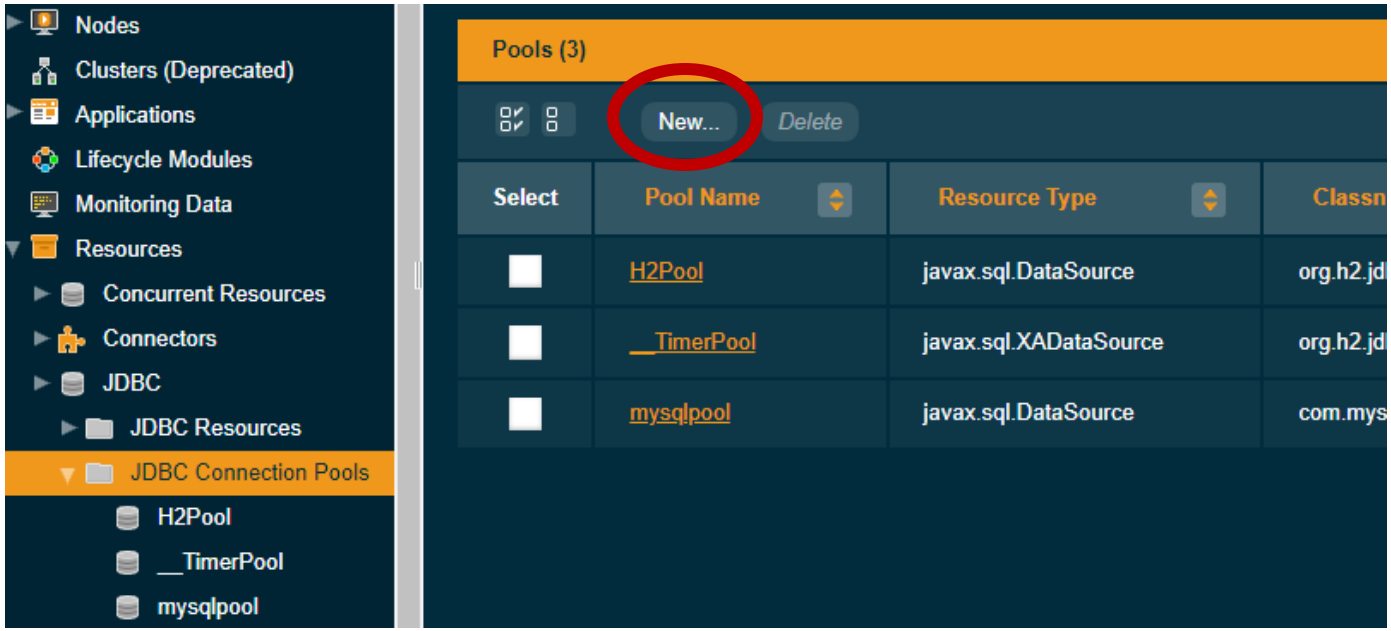
```
Use "exit" to exit and "help" for online help.
asadmin> add-library
Enter the value for the files operand> c:\\users\\o_molloy\\mysql-connector-java-8.0.30.jar
Command add-library executed successfully.
asadmin>
```

It should put the .jar file in

<Payara install directory> \glassfish\domains\domain1\lib

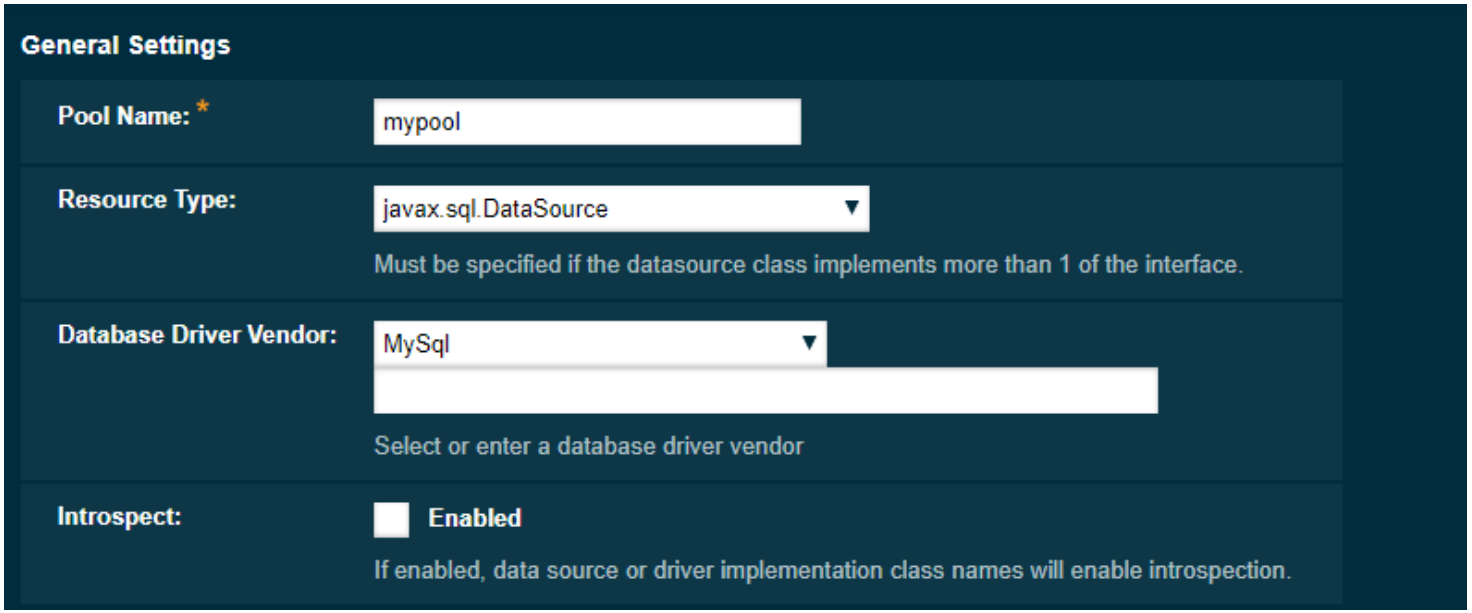
Open Payara admin tool

- Right click on your server in NetBeans and select 'View Domain Admin Console'
- Go down to Resources and drill down to JDBC Connection Pools
- …and select New



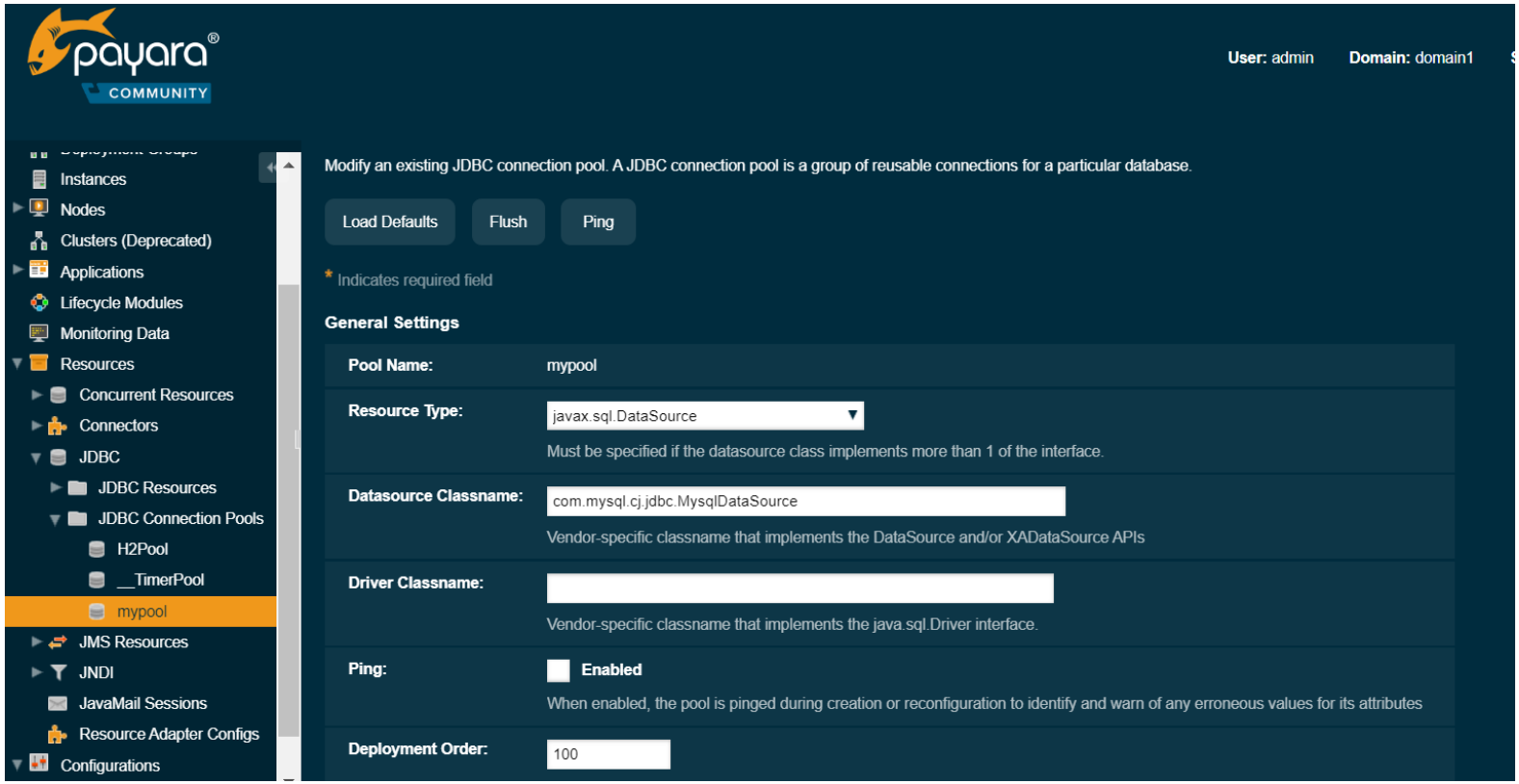
Create new JDBC connection pool

- Pick a simple name
- Resource type: javax.sql.DataSource
- Database Driver Vendor: MySQL



…next

- Replace the Datasource Classname with:
  - com.mysql.cj.jdbc.MysqlDataSource



…next

- Then scroll down to 'Additional Properties'
- Select all properties and delete them. Then add the following properties - using your own values of course!

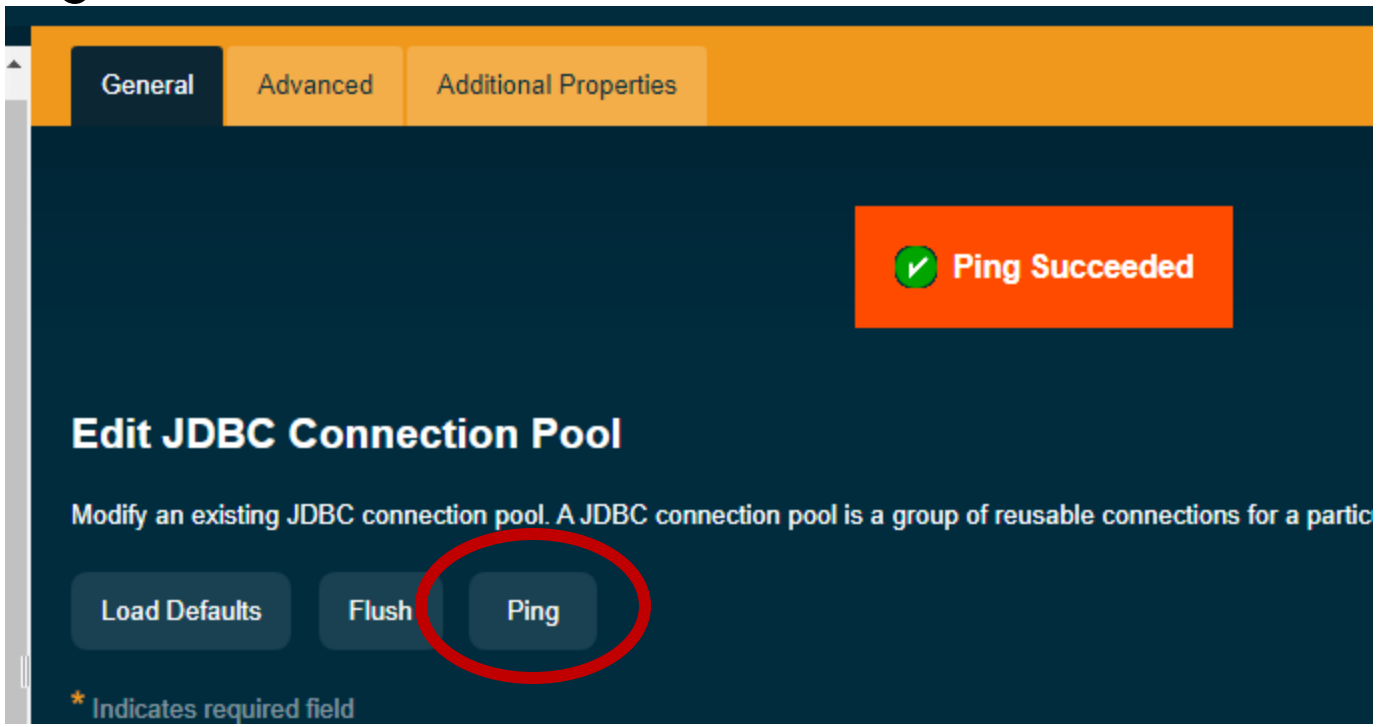
Additional Properties (7)

Add Property | Delete Properties

Select	Name	Value
	useSSL	false
	portNumber	3306
	user	mydb1860mo
	serverName	danu6.it.nuigalway.ie
	databaseName	mydb1860
	password	qo6qop
	driverClass	com.mysql.cj.jdbc.Driver

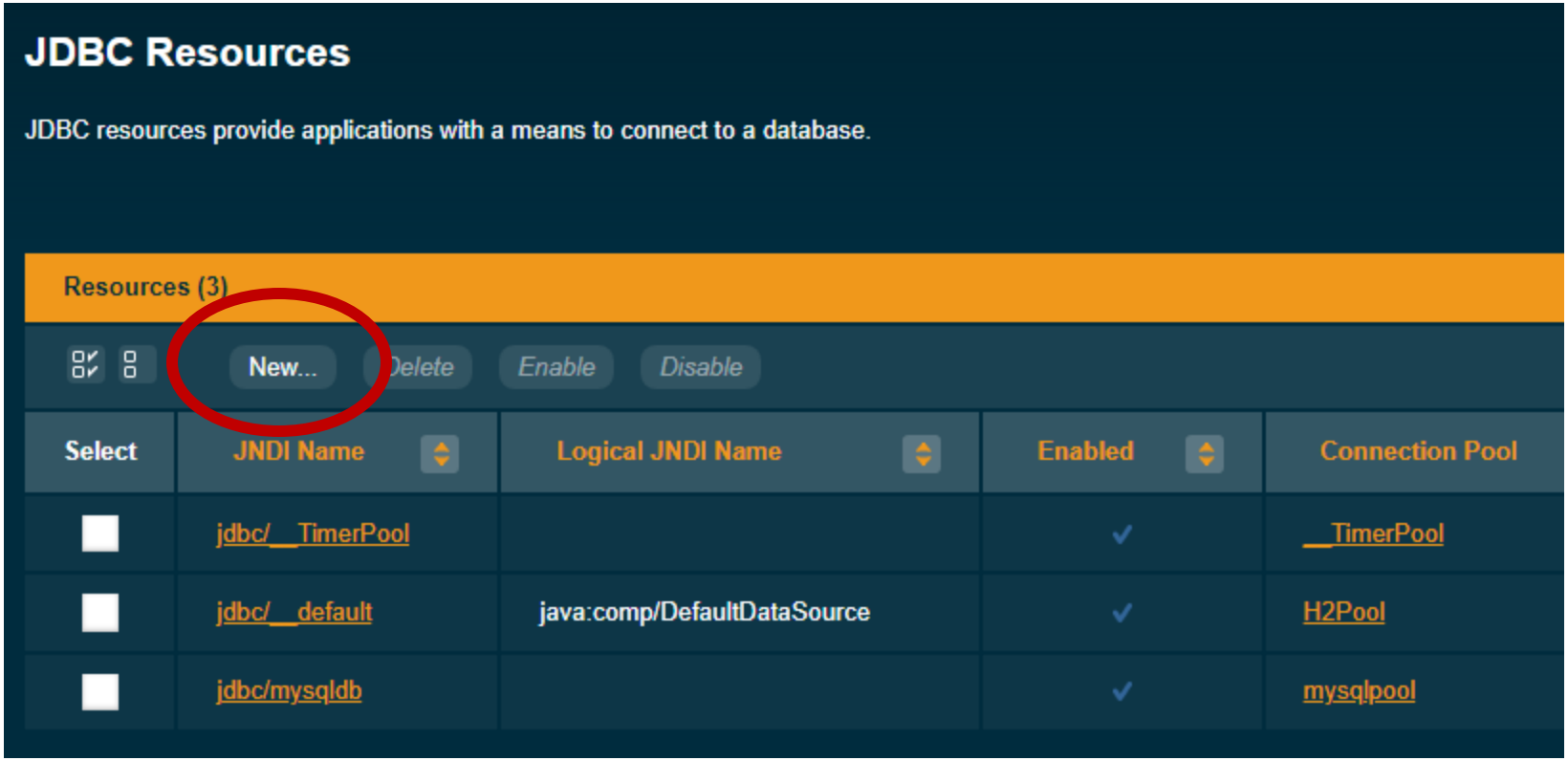
If you are successful

- When you finish you should be able to successfully Ping the database



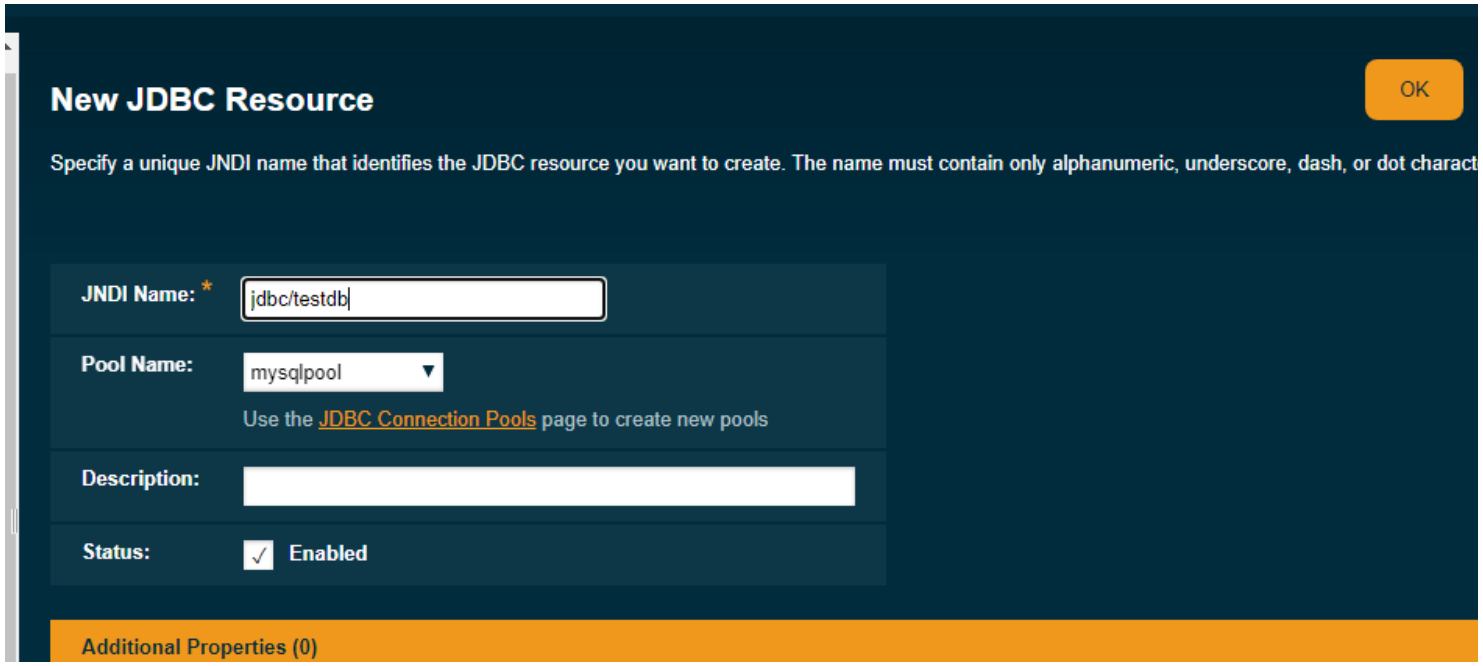
Now create a JDBC resource which uses that connection pool

- Under JDBC Resources select 'New'



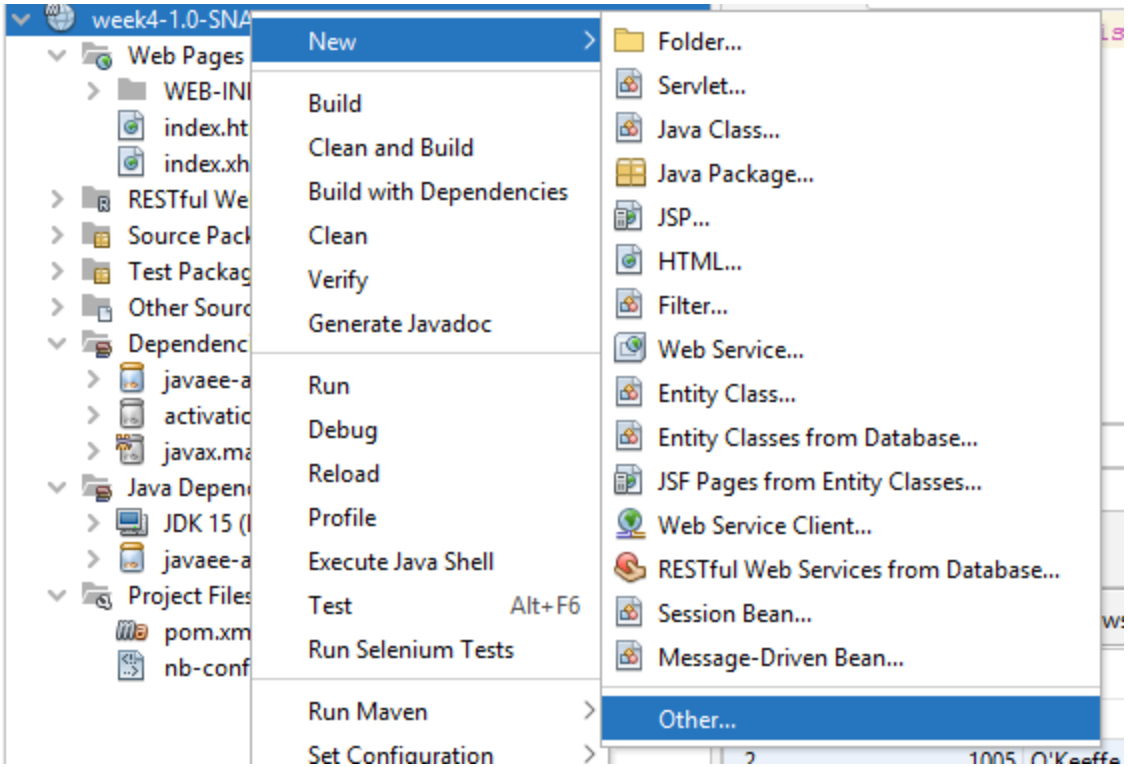
Set up the new JDBC Resource

- Give it a JNDI name: it must be of the form: jdbc/xxx
- Select the pool you have just created
- That's it – select 'OK'



Next Create Persistence Unit

- This is used by the application container to get connections to the database
- Right click on the Project name
- Select New -> Other

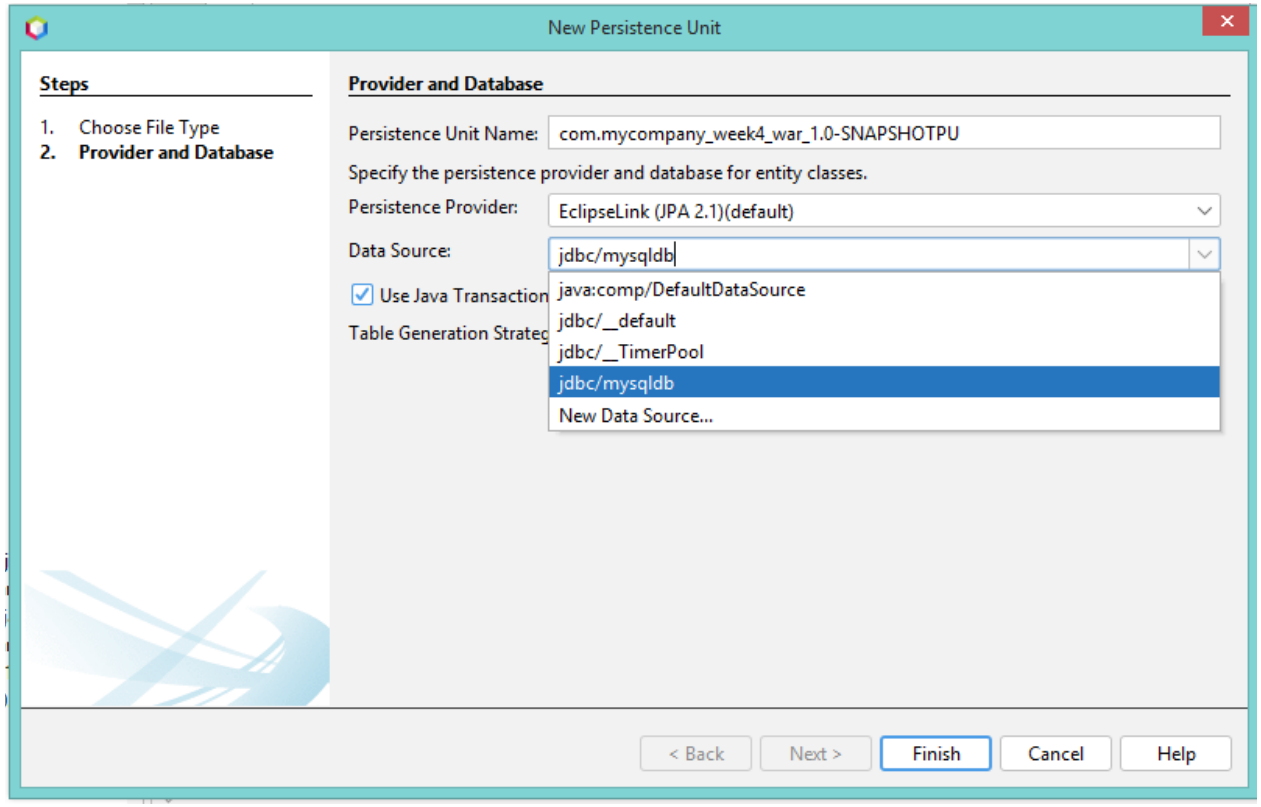


Select Persistence Unit

- The ‘New file’ dialog pops up
- Select ‘Category’ -> ‘Persistence’ and then ‘FileTypes:’ -> ‘Persistence Unit’
- Click on ‘Next’

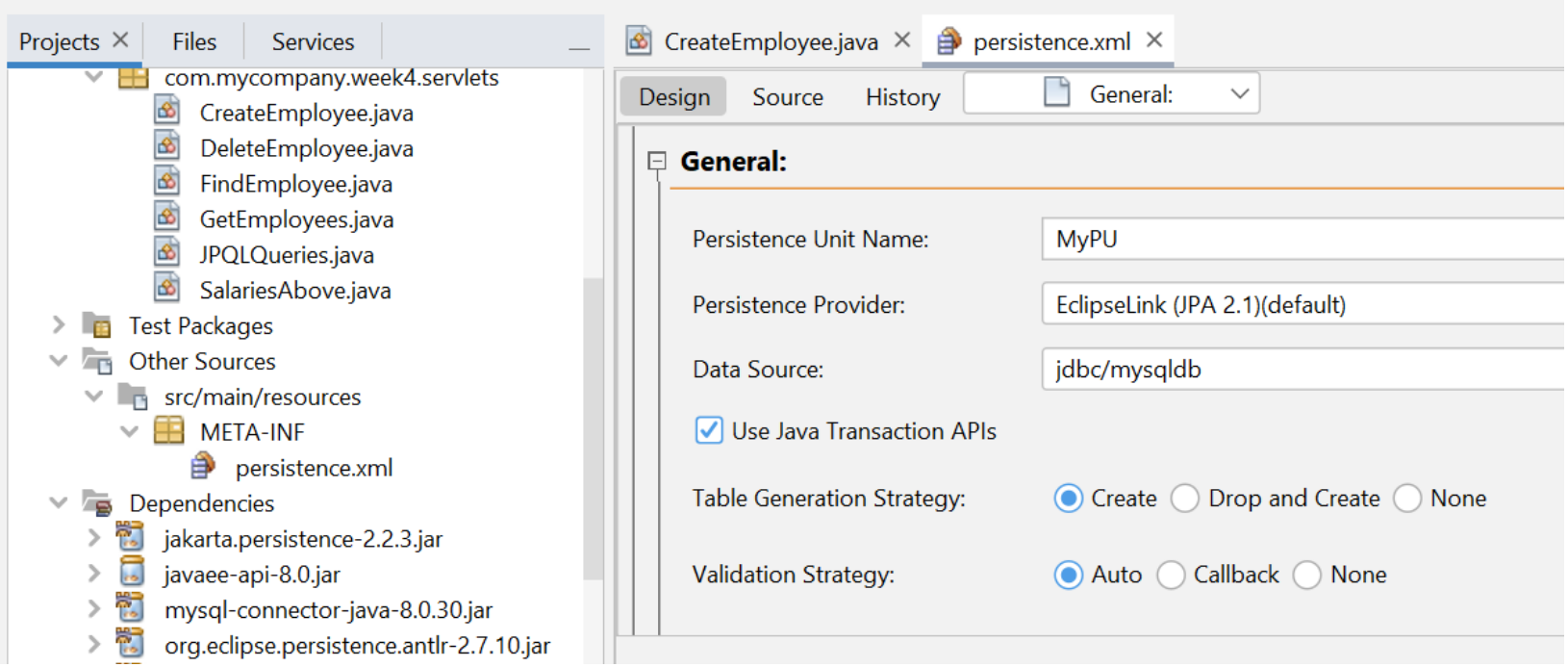
Persistence Unit properties

- You should give the PU (Persistence Unit) a simple name
- Then for the Data Source, select the new JDBC Resource you just created
- And accept the other default settings



Persistence Unit file

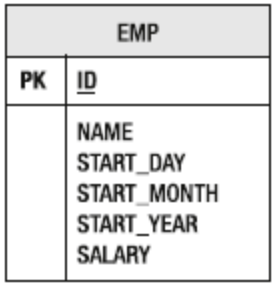
- NetBeans will create a file called ‘persistence.xml’ which contains the information you have entered
- You shouldn’t have to change anything in it for now, so just close it



JPA overview

- Bridging the gap between object-oriented and relational models : ORM (Object-Relational Mapping)
- Used to *persist* our object data in relational form
- Generally 1:1 mapping is not a problem, although you may have to map parts of a Java object to different columns, e.g.

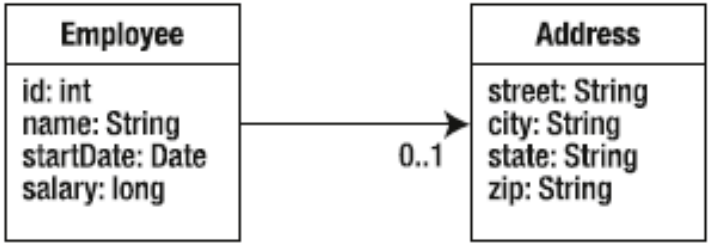
Java



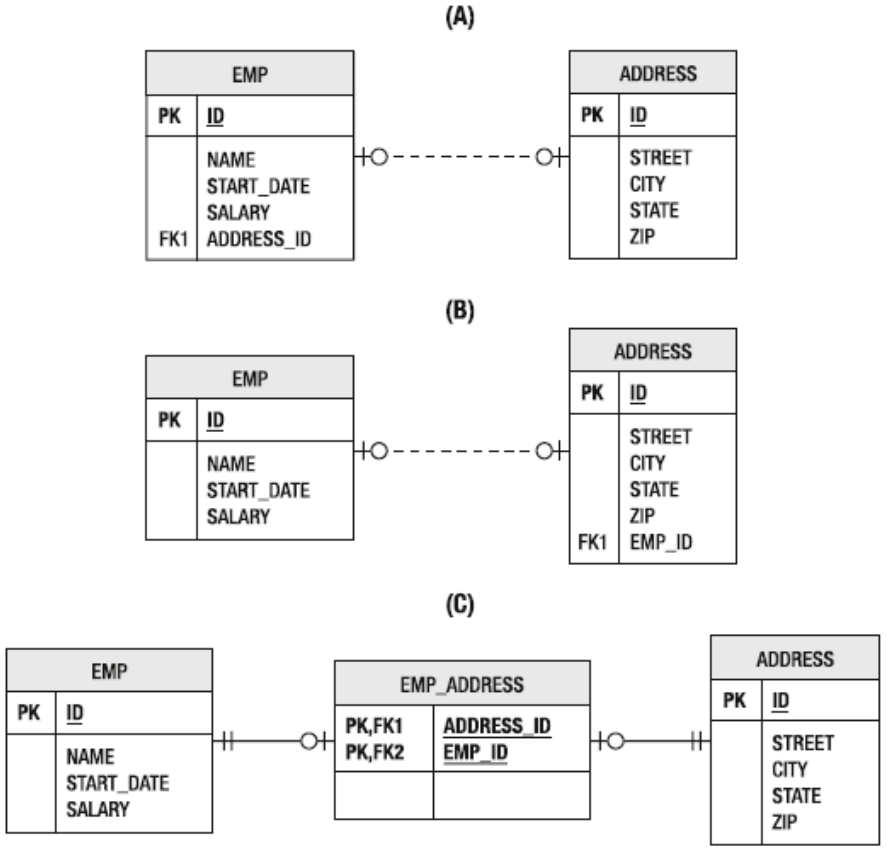
DB

Relations are where it get's tricky

- There can be multiple scenarios for mapping classes to tables or vice-versa
- We may have to introduce PK's or associate classes at either end



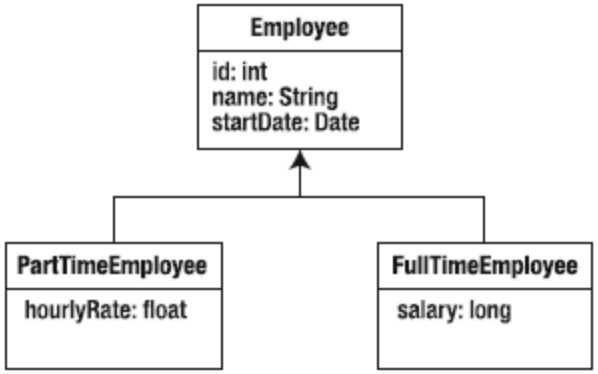
Java

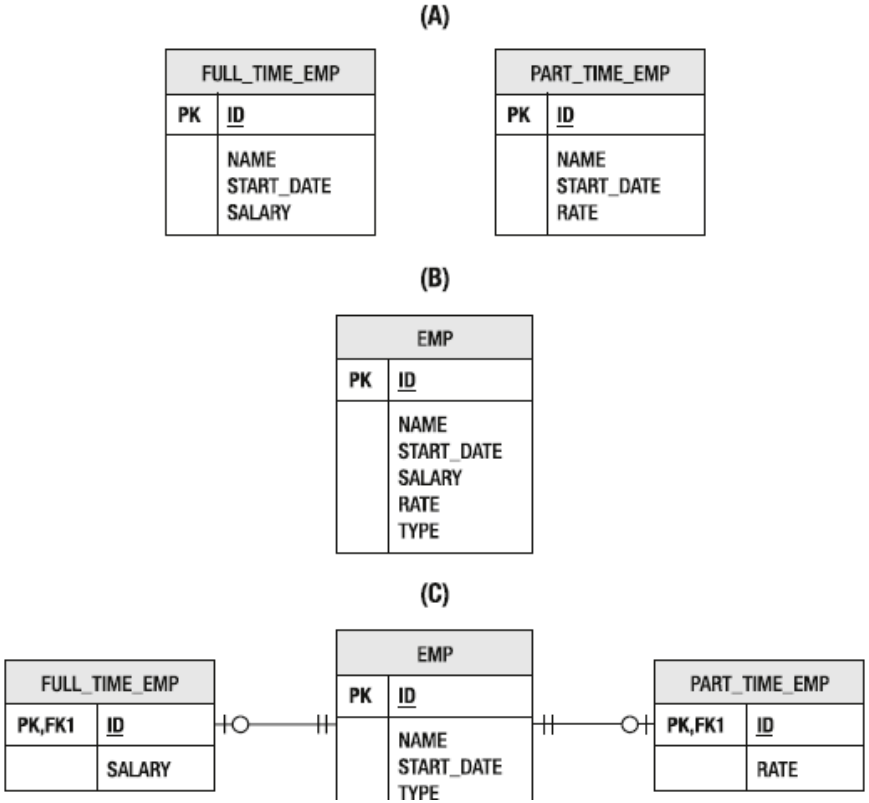


DB

Inheritance also needs handling

- A is the simplest scenario, but queries are separate for emp types
- B is efficient but not normalised, and mapping is more complicated
- C is the likely DB design choice, but requires more complicated classes, queries and additional association class





Classes or Tables first

- You will generally have to deal with both situations
  - Applications for which you generate a new DB schema based on the classes
  - Applications which access existing database (schemas) and where you have to decide how to manage the ORM
- JPA supports (on the Java side) all of the mappings you would expect, e.g.
  - One-to-one
  - One-to-many
  - Many-to-one
  - Many-to-many
- These mappings (and other aspects of the ORM) are defined on the Java side using *annotations*, e.g.
  - @Entity
  - @Table
  - @OneToMany

@Entity

- An entity (from the JPA perspective at least) is an object
  - Is persistable
  - Is unique (must have a primary key / unique id)
  - Transactional (can perform create, update, delete)
  - Granularity (not primitive types)
- Basic requirements to transform Java class into entity
  - No-argument constructor
  - Annotation – at a minimum we need:
    - @Entity, @Id
  - Generally entities don't have to be serialisable, but keys / composite key classes do

Employee entity class

```
@Entity
@Table(name = "Employee")
public class Employee implements Serializable
{
    @Id
    @Column(name="id")
    private int empid;
    @Column(name = "name")
    private String name;
    @Column(name = "salary")
    private long salary;
    public Employee()
    {
    }
    public Employee(int empid, String name, long salary)
    {
        this.empid = empid;
        this.name = name;
        this.salary = salary;
    }
```

Identify the class as a JPA entity

Specify which table to map to

Serializable not strictly speaking necessary but no harm

This is the PK

@Column : Can specify which columns to map to – obviously types must be compatible

@Entity and @Id

Are the minimum requirements for JPA to be able to persist objects

Empty constructor

Also provide other constructor(s)

PLUS getters and setters mandatory

JPA Entity Manager

- We need an entity manager (em), which implements the JPA
  - The entity manager is the interface by which we interact with the Persistence Context (basically a cache within which entities and transactions are managed)
  - The em is used to access the db and run all queries
  - Objects are *managed* by the em
- An Entity Manager Factor (emf) interface is used to provide an em, e.g.

```
EntityManagerFactory emf=Persistence.createEntityManagerFactory(PUame");
```

  - Where PuName is the name of a persistence unit (defined in Persistence.xml)

- Rather than create the emf and em ourselves, though, we can use Context Dependency Injection, where the application container provides and manages the em
  - This just requires adding these lines to the class where you want to use the em:

```
@PersistenceContext(unitName = "MyPU")
private EntityManager em;
```

Benefits of container managed entity manager

- Don't need to open and close the em / emf ourselves
- It provides container-managed transactions (which can span different objects with the application)

A simple example

- Look at the *GetEmployees.java* servlet in the sample code

```
@WebServlet(name = "GetEmployees", urlPatterns = {"/GetEmployees"})

public class GetEmployees extends HttpServlet
{
    @PersistenceContext(unitName = "MyPU")
    private EntityManager em;

protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException
    {
        List<Employee> employees = new ArrayList<>();

        Query q = em.createQuery("select e from Employee e");
        employees = q.getResultList();


        HttpSession session = request.getSession();
        session.setAttribute("employees", employees);
        RequestDispatcher dispatcher =
request.getRequestDispatcher("displayEmployees.jsp");
        dispatcher.forward(request, response);
    }
```

Use a container-managed entity manager

This is a JPA Query, written in JPQL

Inserting an entity

- Look at code in *CreateEmployee.java* servlet
- Some of the more important lines:

```
@PersistenceContext(unitName = "MyPU")
private EntityManager em;
@Resource
private UserTransaction userTransaction;
…
Employee e1 = new Employee(id, name, (long) salary);
userTransaction.begin();
em.persist(e1);
em.flush();
userTransaction.commit();
```

Using a container managed persistence context

Using a container managed transaction

Begin a transaction

Save entity to database

Make sure changes in the persistence context are saved to the DB

Commit the transaction

Running a query

- Just some selected lines from servlet *SalariesAbove.java*

```
@PersistenceContext(unitName = "MyPU")
private EntityManager em;

…
String sthreshold = request.getParameter("threshold");
List<Employee> employees = new ArrayList<>();
Query q = em.createQuery("select e from Employee e where
e.salary > " + sthreshold);

employees = q.getResultList();
```

Create a Query

Run query and get resultset

find

- Used to find an entity given it's primary key
- Sample lines from FindEmployee.java

```
@PersistenceContext(unitName = "MyPU")
private EntityManager em;
...


String id = request.getParameter("id");
int iid = Integer.parseInt(id);


Employee e = em.find(Employee.class, iid);
```

remove

- Like create, update and delete type queries, this must be in a transaction
- Select lines from DeleteEmployee.java:

```
@PersistenceContext(unitName = "MyPU")
private EntityManager em;
@Resource
private UserTransaction userTransaction;
.. .. ..
String sid = request.getParameter("id");
int iid = Integer.parseInt(sid);

userTransaction.begin();
    Employee e = em.find(Employee.class, iid);
    em.remove(e);
    em.flush();
userTransaction.commit();
```

Must (find) bring entity into the persistence context first – i.e. it is then in the 'managed' state

Querying the Persistence Storage

- The Java Persistence query language (JPQL) allows you to perform both dynamic and static queries on the entities in your application.
- The language is like SQL in many ways. However, it does have benefits over SQL. The Java Persistence query language operates over the entities and their relationships rather than over the actual relational database schema. This makes queries portable regardless of the underlying database.
- Queries come in three different flavours: select, update, and delete.
  - A select query returns a set of entities from your database. The set usually has specific constraints that limit the result set.
  - An update query changes one or more properties of an existing entity or set of entities.
  - A delete statement removes one or more entities from the database.

Select

- You have several options to create a query. The most basic way is to simply ask the entity manager for one. The select query applies your specific criteria when it retrieves entities.

```
Query q1 = em.createQuery("select e from Employee e where e.name = 'mary'");
```

- you may want to programmatically set the parameters of the where clause. You can do that by calling the query object's *setParameter* method when it has parameterized elements. The following code creates the same query and prints the results, but it allows you to dynamically set the name:

```
Query q2 = em.createQuery("select e from Employee e where e.name = :name");
q2.setParameter("name", "mary");
Employee e2 = (Employee) q2.getSingleResult();
```

Update

- One you've retrieved a managed entity, either by querying the database with the query language or by using the find method, updating the entity is as easy as modifying its properties and committing the open transaction.

```
userTransaction.begin();
{
    e2.setSalary((long) 450000.00);
    em.persist(e2);
}
userTransaction.commit();
```

More sample queries

- From JPQLQueries.java

- No transaction needed for straight query without change to DB

```
// Select Query
Query q1 = em.createQuery("select e from Employee e where e.name = 'mary'");
Employee e1 = (Employee) q1.getSingleResult();
System.out.println("Employee with name mary has id: " + e1.getEmpid());
System.out.println();
```

- Using parameterised elements allows us to easily insert data values into queries

```
// Select Query with parameterised elements
Query q2 = em.createQuery("select e from Employee e where e.name = :name");
q2.setParameter("name", "Marg");
Employee e2 = (Employee) q2.getSingleResult();
System.out.println("Employee with name Marg has id: " + e2.getEmpid());
System.out.println();
```

- To save changes to an entity which is already 'managed' (has already been retrieve / created using the entity manager) use the merge() method

```
// Update
e2.setSalary((long) 450000.00);
userTransaction.begin();
{
   em.merge(e2);
   em.flush();
}
userTransaction.commit();
```

Named Queries

- Named queries are different from dynamic queries in that they are static and unchangeable.
- In addition to their static nature, which does not allow the flexibility of a dynamic query, named queries can be more efficient to execute because the persistence provider can translate the JPQL string to SQL once the application starts, rather than every time the query is executed.
- Named queries are static queries expressed in metadata inside either a @NamedQuery annotation or the XML equivalent.
- To define these reusable queries, annotate an entity with the @NamedQuery annotation, which takes two elements: the name of the query and its content.

The Customer Entity Defining Named Queries


```
@Entity
@NamedQueries({
  @NamedQuery(name = "findAll", query="select c from Customer c"),
  @NamedQuery(name = "findVincent", query="select c from Customer c where c.firstName = 'Vincent'"),
  @NamedQuery(name = "findWithParam", query="select c from Customer c where c.firstName = :fname")
})
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private Integer age;
    private String email;
    @OneToOne
    @JoinColumn(name = "address_fk")
    private Address address;
    // Constructors, getters, setters
}
```

Named Queries (continue)

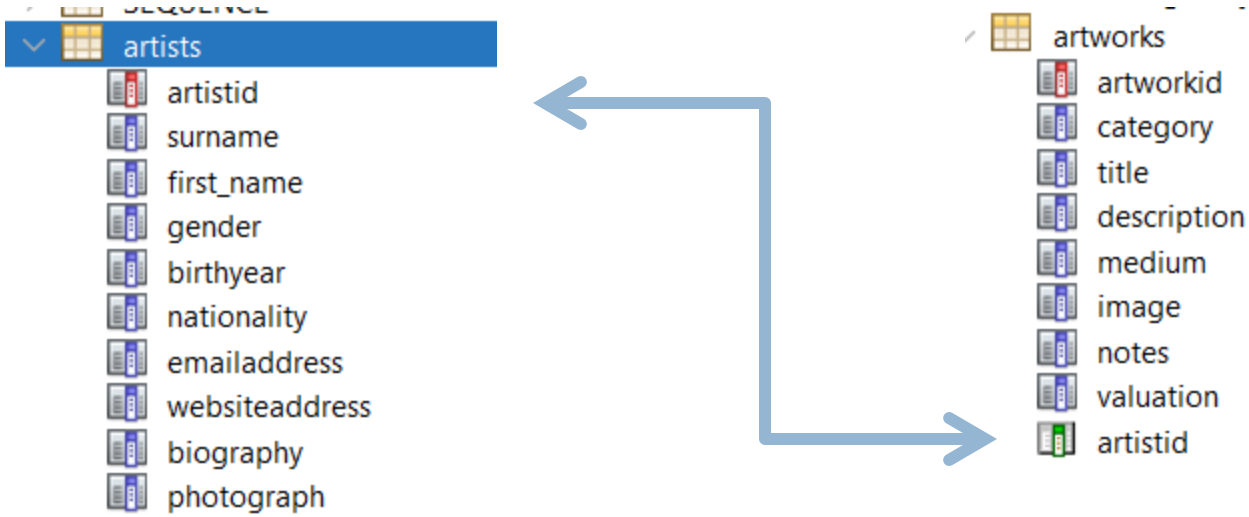
- The way to execute these named queries resembles the way dynamic queries are used.
- The EntityManager.createNamedQuery() method is invoked and passed to the query name defined by the annotations.
- This method returns a Query that can be used to set parameters, the max results, fetch modes, and so on.
- To execute named queries:

```
Query query = em.createNamedQuery("findAll");
List<Customer> customers = query.getResultList();
```

```
Query query = em.createNamedQuery("findWithParam");
query.setParameter("fname", "Vincent");
query.setMaxResults(3);
List<Customer> customers = query.getResultList();
```

Entity Relationship mapping

- Example where I have a table **artworks**, with a single foreign key, referencing the table **artists**



JPA mapping on the **artists** side

- Need a Collection to hold the artworks
- Specifying the name of the Java property used to reference this Artist object on the other side of the relationship in Artworks

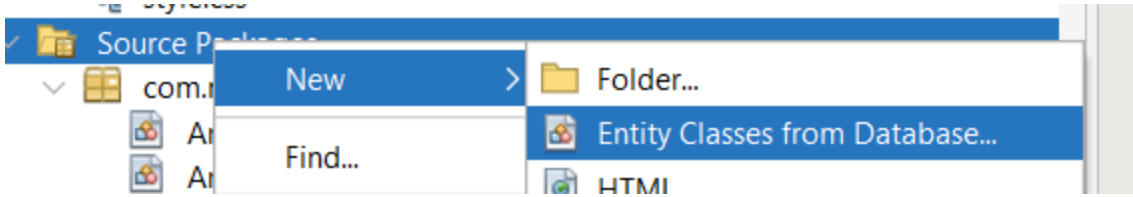
```
@OneToMany(mappedBy = "artistid")
private Collection<Artworks> artworksCollection;
```

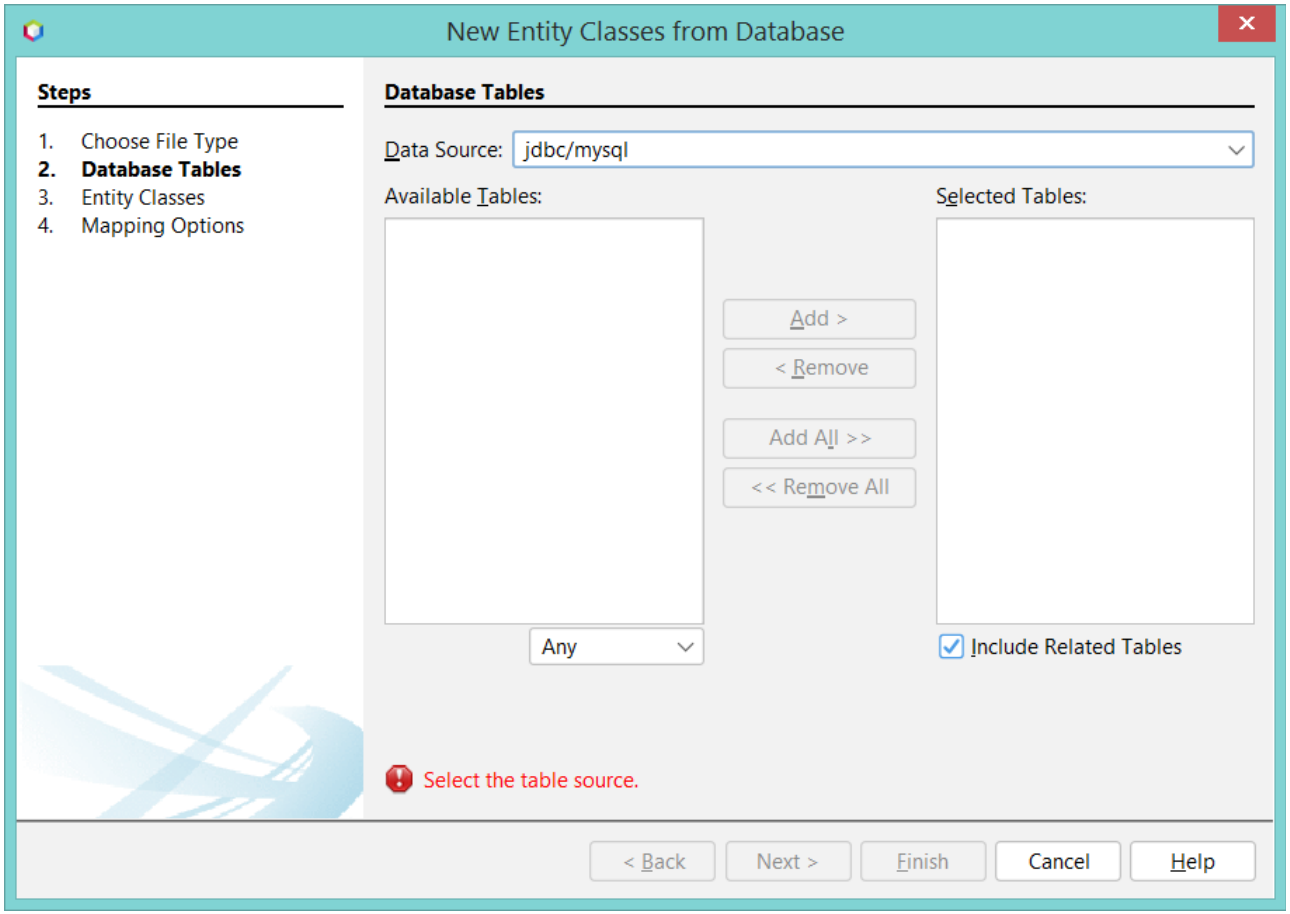
JPA mapping on the **artworks** side

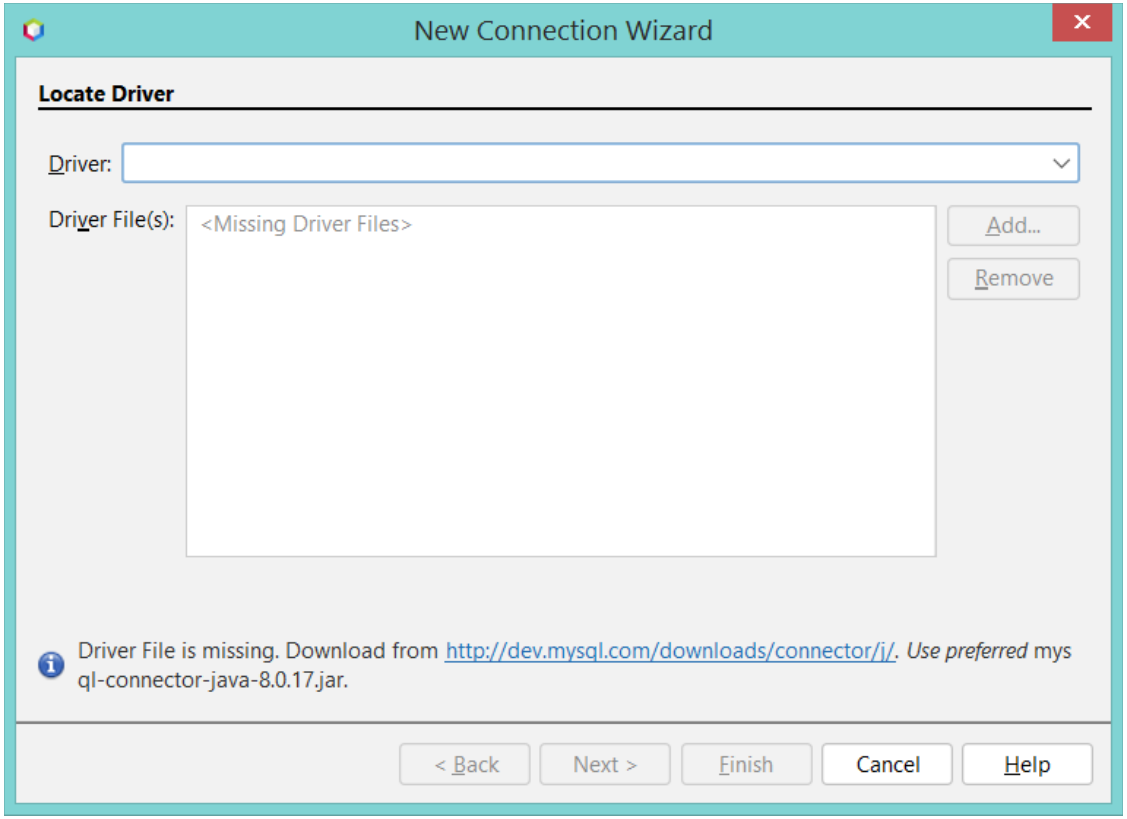
- Just need to reference a single Artist object
- Specifying the name of the property and column to map to in the Artist object

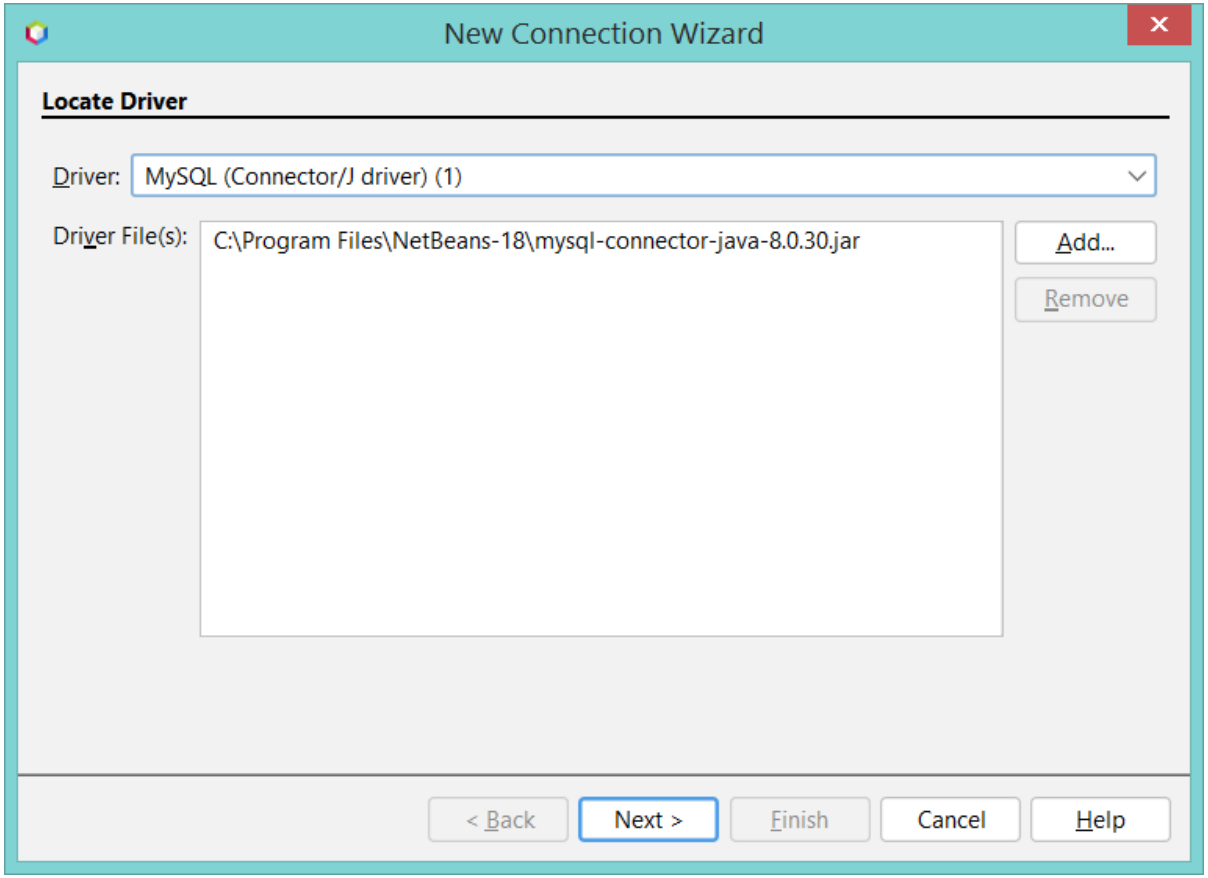
```
@JoinColumn(name = "artistid", referencedColumnName = "artistid")
@ManyToOne
private Artists artistid;
```

Autogenerate classes using NetBeans

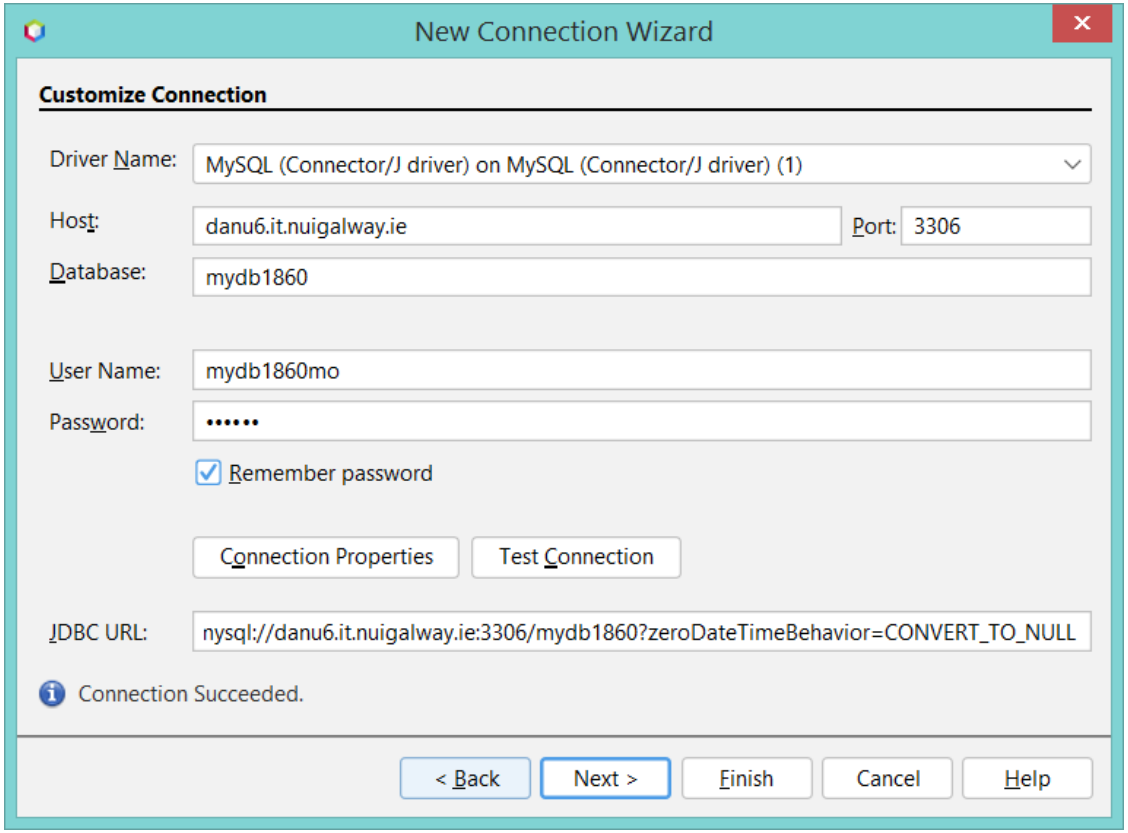



New Entity Classes from Database
Steps
1. Choose File Type
2. Database Tables
3. Entity Classes
4. Mapping Options
Database Tables
Data Source: jdbc/mysql
Available Tables:
Selected Tables:
Add >
< Remove
Add All >>
<< Remove All
Any
Include Related Tables
Select the table source.
< Back
Next >
Finish
Cancel
Help

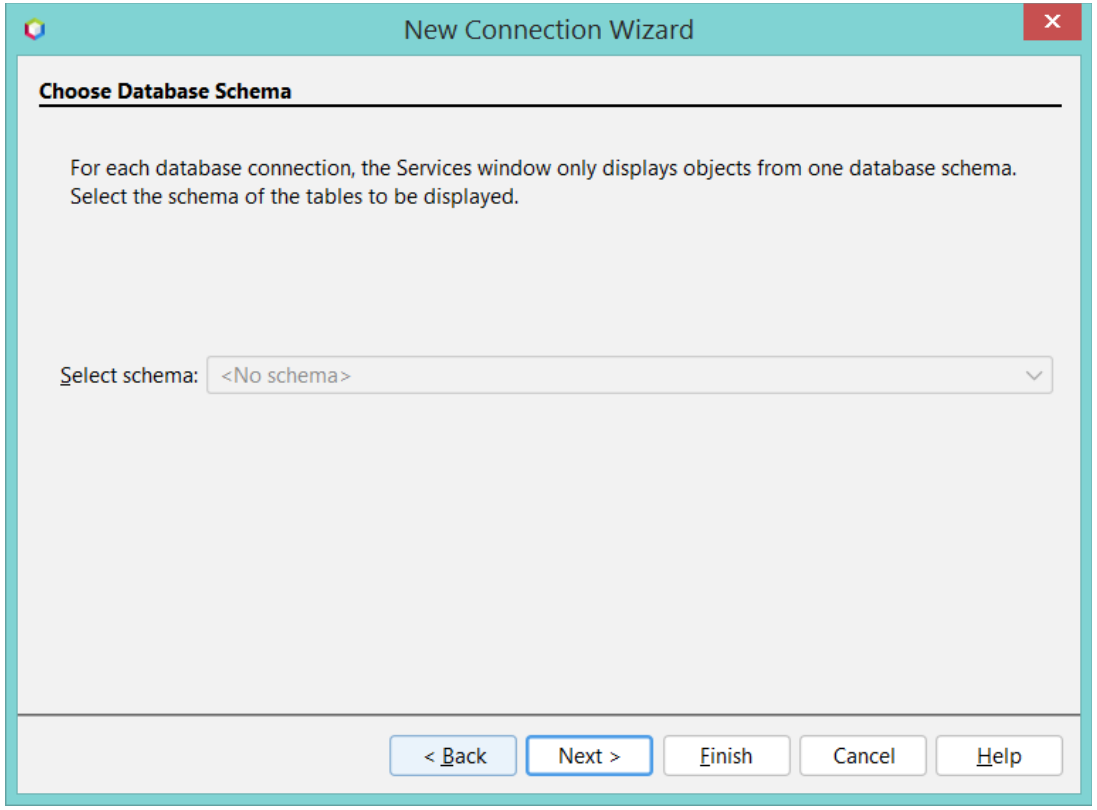

New Connection Wizard
Locate Driver
Driver:
Driver File(s):
<Missing Driver Files>
Add...
Remove
Driver File is missing. Download from http://dev.mysql.com/downloads/connector/j/. Use preferred mysql-connector-java-8.0.17.jar.
< Back
Next >
Finish
Cancel
Help


New Connection Wizard
Locate Driver
Driver:
MySQL (Connector/J driver) (1)
Driver File(s):
C:\Program Files\NetBeans-18\mysql-connector-java-8.0.30.jar
Add...
Remove
< Back
Next >
Finish
Cancel
Help

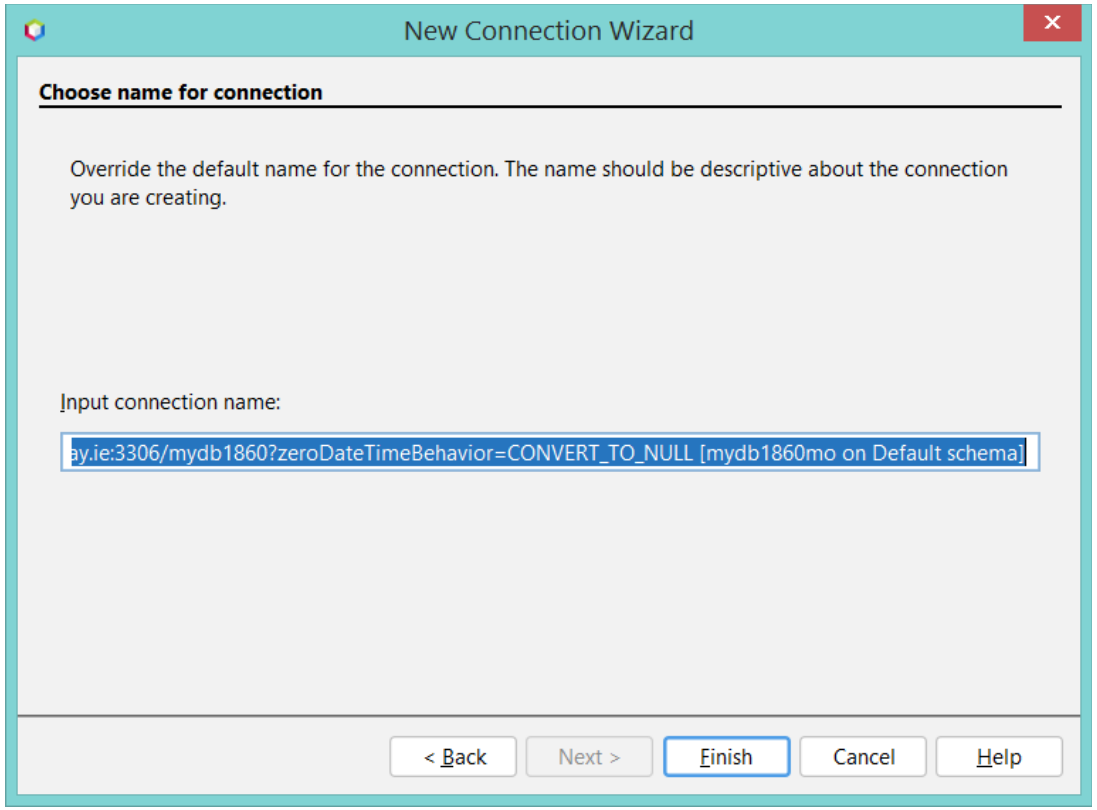
- Need to fill in all these correctly

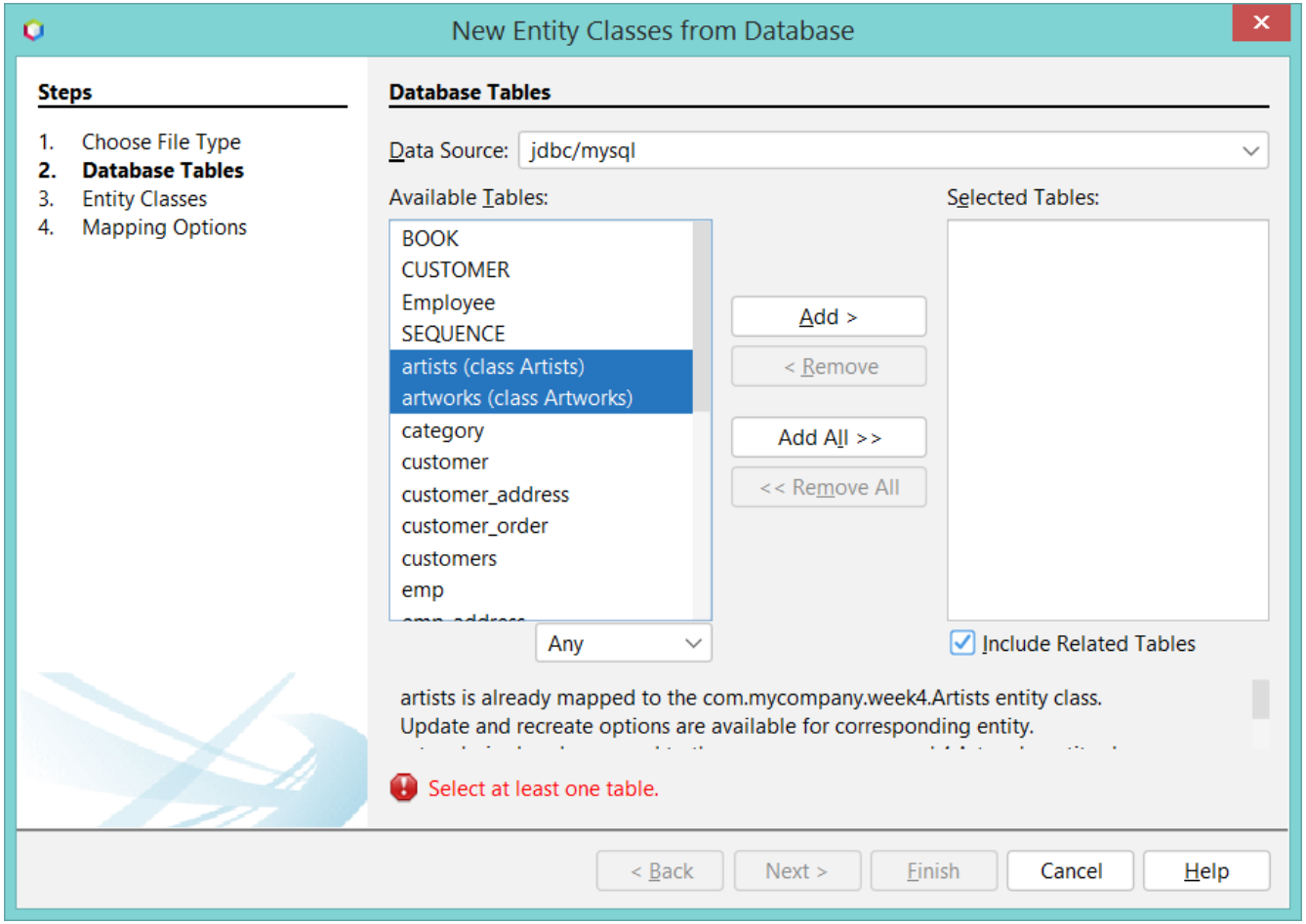


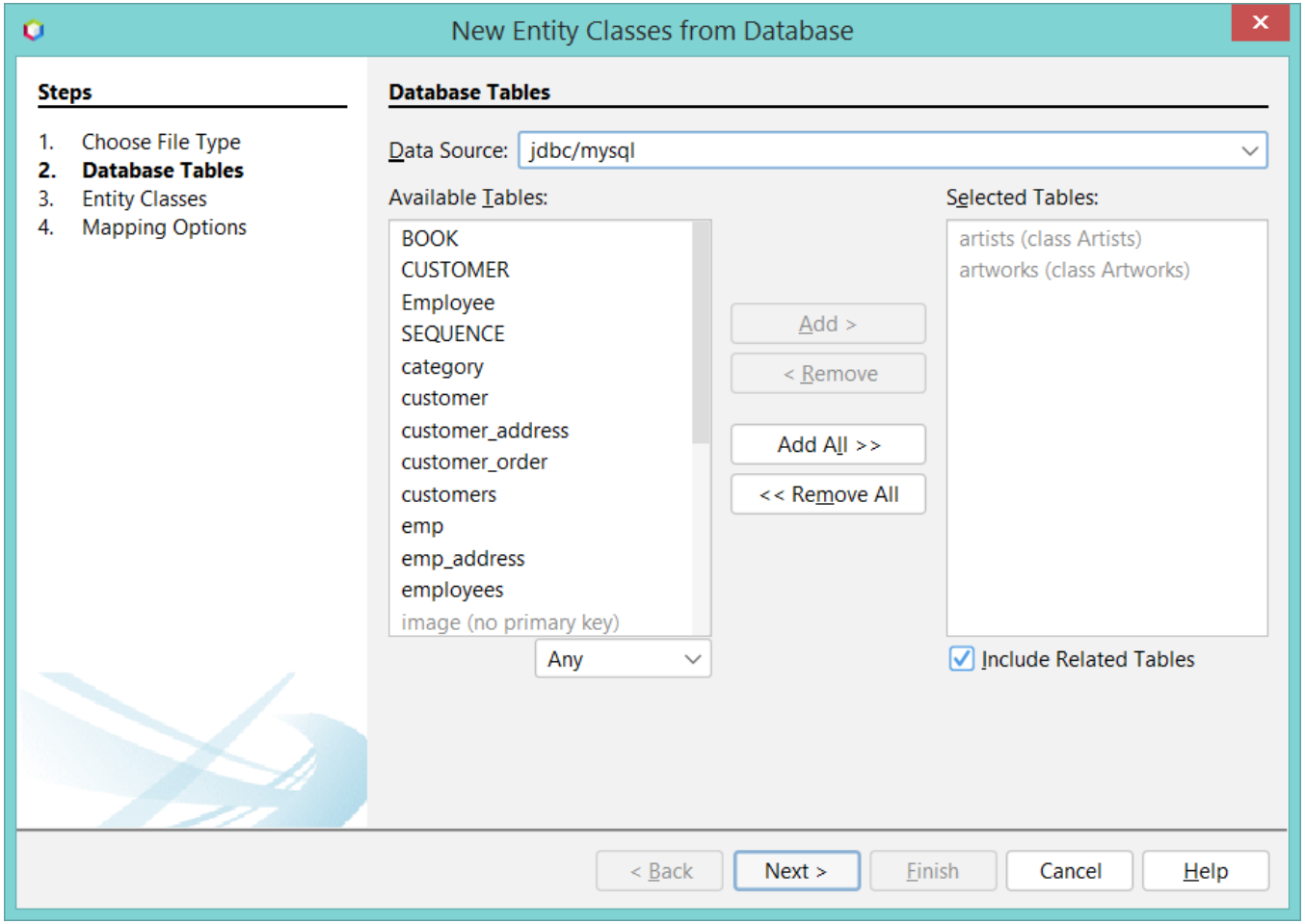
- Pick schema if there is one

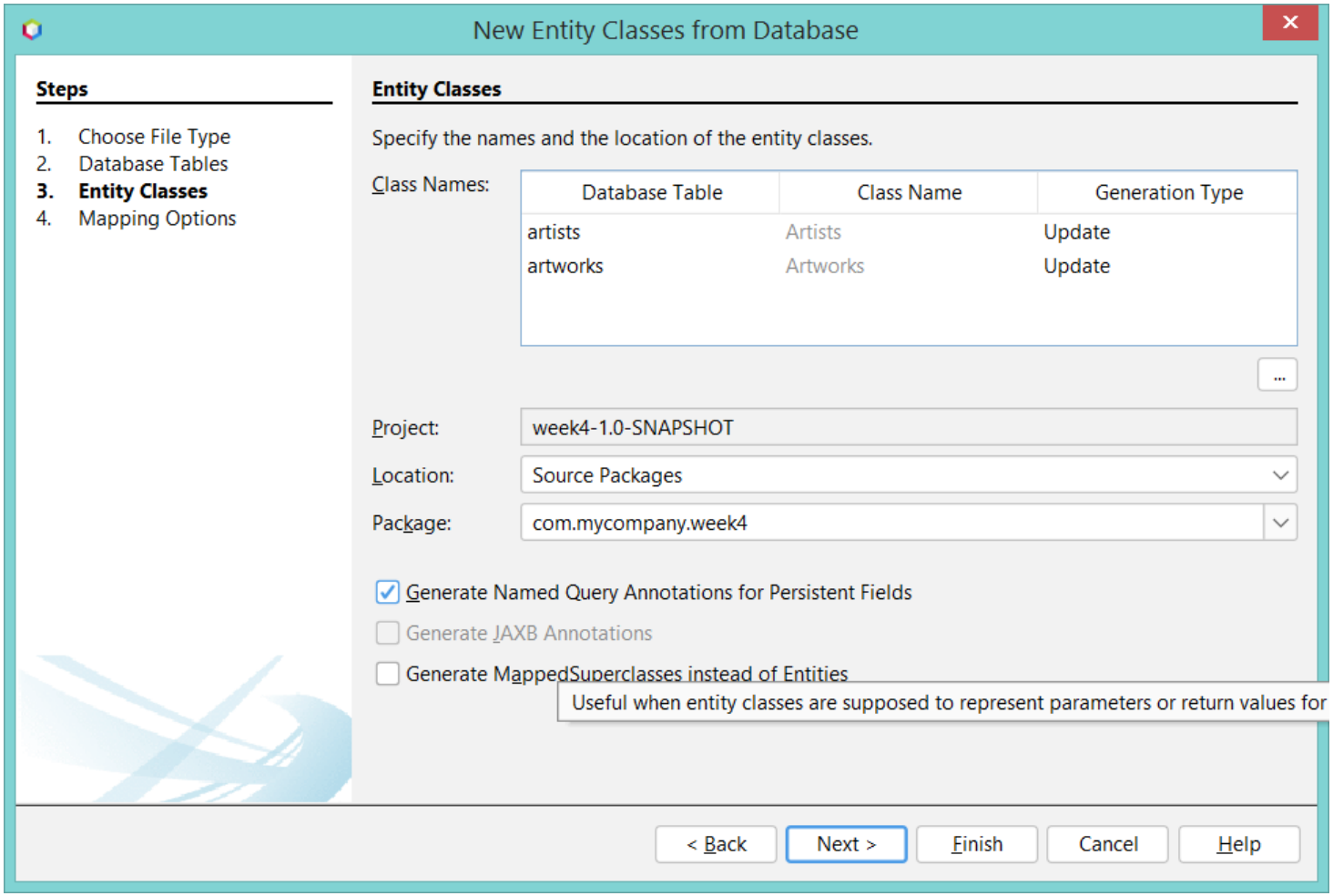


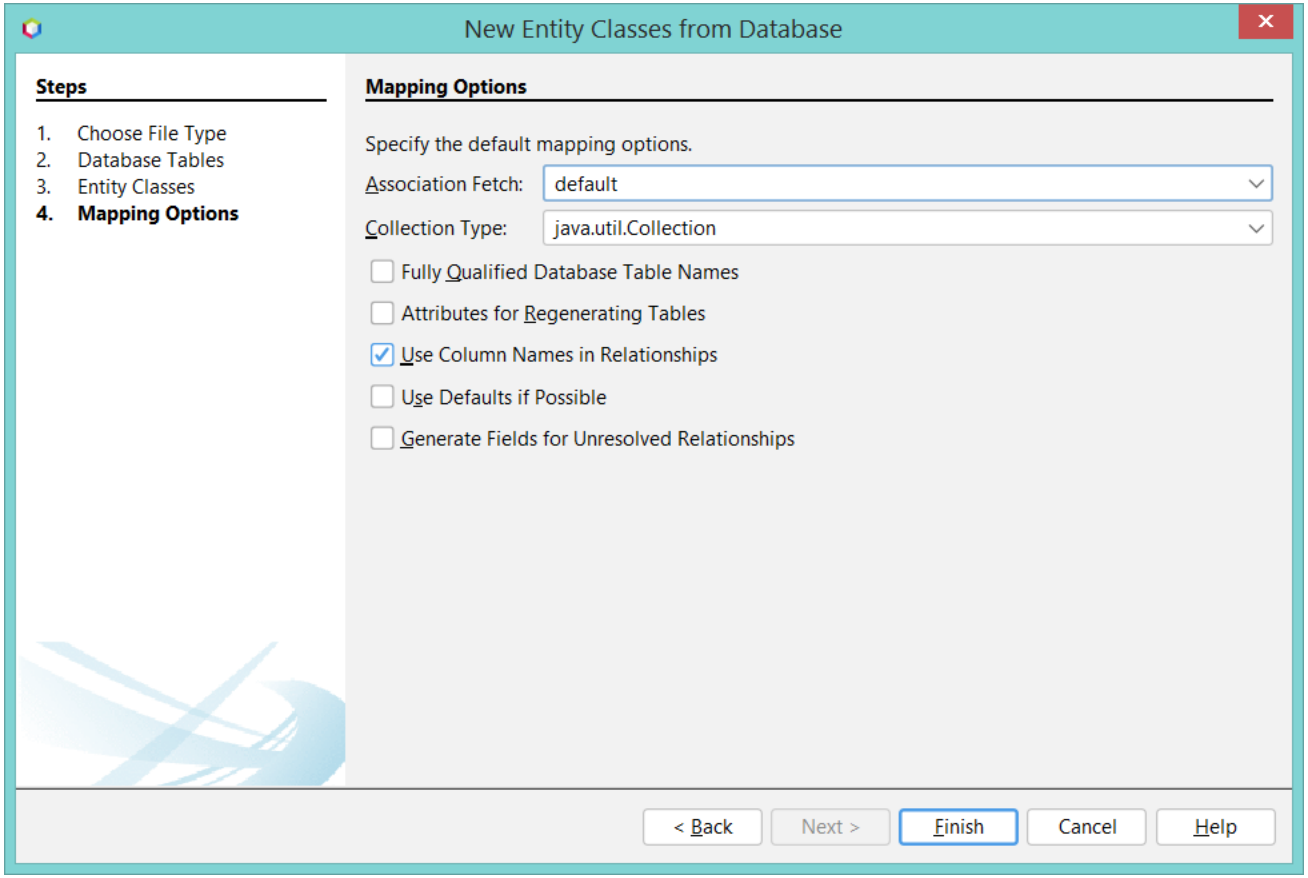
- Accept default here




New Entity Classes from Database
Steps
1. Choose File Type
2. Database Tables
3. Entity Classes
4. Mapping Options
Database Tables
Data Source: jdbc/mysql
Available Tables:
BOOK
CUSTOMER
Employee
SEQUENCE
artists (class Artists)
artworks (class Artworks)
category
customer
customer_address
customer_order
customers
emp
Add >
< Remove
Add All >>
<< Remove All
Selected Tables:
Any
Include Related Tables
artists is already mapped to the com.mycompany.week4.Artists entity class.
Update and recreate options are available for corresponding entity.
Select at least one table.
< Back
Next >
Finish
Cancel
Help


New Entity Classes from Database
Steps
1. Choose File Type
2. Database Tables
3. Entity Classes
4. Mapping Options
Database Tables
Data Source: jdbc/mysql
Available Tables:
BOOK
CUSTOMER
Employee
SEQUENCE
category
customer
customer_address
customer_order
customers
emp
emp_address
employees
image (no primary key)
Any
Add >
< Remove
Add All >>
<< Remove All
Selected Tables:
artists (class Artists)
artworks (class Artworks)
Include Related Tables
< Back
Next >
Finish
Cancel
Help

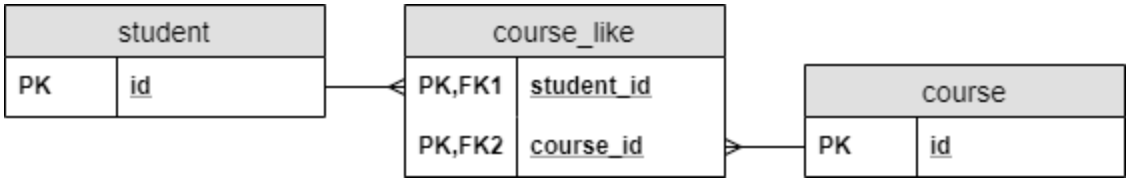

New Entity Classes from Database
Steps
1. Choose File Type
2. Database Tables
3. Entity Classes
4. Mapping Options
Entity Classes
Specify the names and the location of the entity classes.
Class Names:
Database Table
Class Name
Generation Type
artists
Artists
Update
artworks
Artworks
Update
...
Project:
week4-1.0-SNAPSHOT
Location:
Source Packages
Package:
com.mycompany.week4
Generate Named Query Annotations for Persistent Fields
Generate JAXB Annotations
Generate MappedSuperclasses instead of Entities
Useful when entity classes are supposed to represent parameters or return values for
< Back
Next >
Finish
Cancel
Help


New Entity Classes from Database
Steps
1. Choose File Type
2. Database Tables
3. Entity Classes
4. Mapping Options
Mapping Options
Specify the default mapping options.
Association Fetch:
default
Collection Type:
java.util.Collection
Fully Qualified Database Table Names
Attributes for Regenerating Tables
Use Column Names in Relationships
Use Defaults if Possible
Generate Fields for Unresolved Relationships
< Back
Next >
Finish
Cancel
Help

Some other examples

- Many to many

- Since both sides should be able to reference the other, **we need to create a separate table to hold the foreign keys**



- In such a join table, the combination of the foreign keys will be its composite primary key

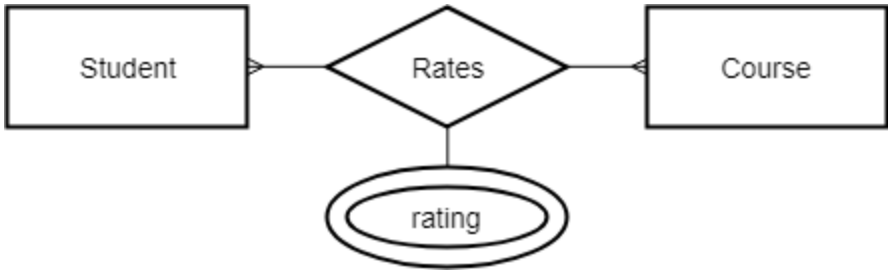
https://www.baeldung.com/jpa-many-to-many

```java
@Entity class Student
{
  @Id
  Long id;

  @ManyToMany
  @JoinTable( name = "course_like",
  joinColumns = @JoinColumn(name =
  "student_id"), inverseJoinColumns
  = @JoinColumn(name = "course_id"))
  Set<Course> likedCourses;


// additional properties
// standard constructors, getters,
and setters
}
```

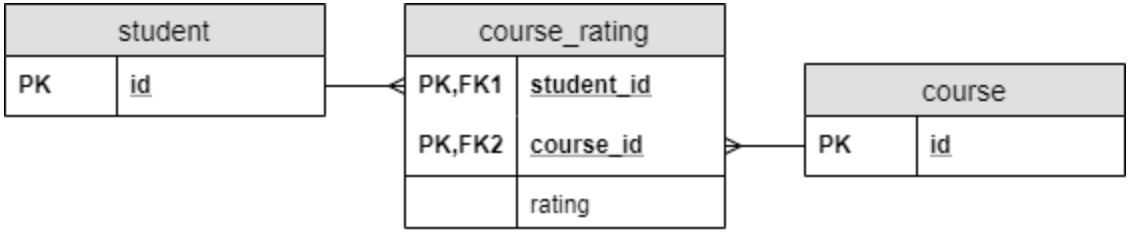
```java
@Entity class Course
{
  @Id
  Long id;

  @ManyToMany (mappedBy =
"likedCourses")
    Set<Student> likes;

// additional properties
// standard constructors, getters,
and setters
}
```

Using a composite key

- when the relationship itself has an attribute



- Need another table



- Need to create a composite (primary) key class

```
@Embeddable class CourseRatingKey implements Serializable
{
  @Column(name = "student_id")
  Long studentId;

  @Column(name = "course_id")
  Long courseId;

  // standard constructors, getters, and setters
  // hashcode and equals implementation
}
```

Then the entity class itself

```
@Entity class CourseRating
{
  @EmbeddedId
  CourseRatingKey id;

  @ManyToOne
  @MapsId("studentId")
  @JoinColumn(name = "student_id")
  Student student;

  @ManyToOne
  @MapsId("courseId")
  @JoinColumn(name = "course_id")
  Course course;

  int rating;

  // standard constructors, getters, and setters
}
```

```java
@Entity class Student
{
  @Id
  Long id;

  @OneToMany(mappedBy = "student")
  Set<CourseRating> ratings;

// additional properties
// standard constructors, getters,
and setters
}
```

```java
@Entity class Course
{
  @Id
  Long id;

  @OneToMany(mappedBy = "course")
  Set<CourseRating> ratings;


  // additional properties
  // standard constructors, getters,
and setters
}
```