A.Y. 2020-2021

Politecnico di Milano, Software Engineering 2 project

POLITECNICO
MILANO 1863

ITD

Implementation & Testing Document

version 1.0

Cirino Duilio - 968466

Cocchia Lorenzo - 968139

Table of contents

1. INTRODUCTION

This document aim is to provide documentation of the project uploaded in the git repository. This has to be considered as the implementation part of the already described Requirements and Analysis of Specification Document and Design Document. Indeed, to give an overview about the specification and to the design adopted it is preferable to pay close attention to those documents.

1.1. Scope

To give an overview of the scope of the project given the specifications, we refer to RASD and DD. The scope of this part is to implement the software described as software-to-be in the yet mentioned documents, with the unique limitation to implement the line-up system and not the book-a-visit one due to constraints on the assignment of the project. For the definition of line-up, again, we refer to that given in the RASD. Moreover, the implementation has to produce only a prototype of the product so some functionality and better-to-have features could be not implemented.

1.2. Document structure

This document is divided into 8 sections.
In the first section an introduction of the document is provided.
In the second section there are described which are the requirements implemented through the project and which are not, with an explanation for all of them.
In the third paragraph the reader can find useful information about adopted development frameworks such as programming languages, middlewares and APIs used to implement the project.
The fourth paragraph serves as a description of the structure used for the code.
The fifth paragraph is about all that concerns testing.
The sixth paragraph provides some useful instructions to install the software produced.
The seventh paragraph describes carefully how much effort has been spent for the development of this project and the relative documentation.

In the eighth paragraph the reader can find some link to documents used as reference for the purpose of the project.

2. REQUIREMENTS IMPLEMENTATION

The first decision we made was to implement only the web application, not the native mobile one expected from the DD. This is because on one hand the team had not sufficient experience with the technology required to implement the mobile app, on the other the functionalities exposed by both webapp and mobile app had to be the same according to the DD. So, after done this decision, we chose to implement some of the goals yet exposed in the RASD and here reproposed: in the following table the reader can find a description of the goal and if it is implemented and eventually why so. If the response is yes, we always intend the goal as a core goal of the application.

Goal number	Description	Implemented
G1.1	allow customers to “line-up” remotely effectively	Yes
G1.2	allow customers to know in real-time the current line-up number	No, the line-up number in the context of the only line-up may be out of context. Accesses to the store are automatically computed by the system.
G1.3	allow customers to see the available spots in the registered supermarkets	Yes, even if this goal was more related to the book-a-visit feature, the grocery is basically always available to line-ups except when the grocery is closed.
G1.4	allow customers to “book a visit” to the store	No, it is not requested by the assignment

G1.5	provide an estimation of the waiting time	Yes
G1.6	provide alarms to customers basing on their position and the position of the store	No, this goal was more intended to be a feature to be applied to the mobile app. Even if this could be possible in the webapp, we did not retain the importance comparable to the one that it would have had in the mobile app.
G1.7	provide fallback options for people who do not have the access to technologies	Yes
G1.8	allow customers to get into the store when they are allowed to	Yes
G1.9	allow users to register to the system	Yes
G1.10	allow users to access to the system with their credentials	Yes
G2.1	allow managers to register and manage a new store	Yes
G2.2	allow managers to monitor the entrance	Yes, this has to be intended as a statistics service which retrieves the number of users in a certain week or month,

		or the average time spent into the grocery in a certain week or a certain month.
G2.3	allow managers to handle the line	Yes
G2.4	allow managers to manage employees	Yes
G2.5	allow employees to manage the lines	Yes

So, summarizing, we decided to implement the following goals:

- G1.1, G1.3, G1.5, G1.7, G1.8, G1.9. G1.10
- G2.1, G2.2, G2.3, G.24, G2.5

For each goal implemented, we decided to implement some of the requirements related to it, here we refresh their description and explain if they are implemented. The main reason why we decided to not implement some requirements was due to the fact that we considered those requirements as nice-to-have requirements and not core requirements. Attached to the table, the reader can find a column that represents the module in which the requirement is mostly implemented. For more information about the mapping between the modules and the requirement our advice is to take another look to the mapping table provided in the Requirements Traceability section on the DD [4.B].

Requirement		Implemented	Reference module
R1	The customer must be able to add himself to the current line	Yes	Queue Update
R2	The customer must be able to provide at least a telephone number	Yes	Login
R3	The customer must be able to search for specific groceries	Yes	Grocery Handler
R4	The customer must be able to do the research basing on his position	Yes	Search Engine

R5	The customer must be able to make the research basing on a position provided by himself	Yes, even if the natural approach is to search basing on the actual position mined by the browser.	---
R6	When a line-up request concludes successfully, an access code has to be provided to the customer	Yes, the access code has to be intended as the ID of the reservation itself in the prototype	Reservation Handler
R10	The customer must be able to know all the available booking spots in every registered grocery store	Yes, as already said, if the grocery is open than it is available actually	Queue Update
R11	Registered customers must be able to access to shortcuts in order to see their previous groceries choices	Yes	Search Engine
R15	The customer must be able to know the estimated time of his visit: both for the line-up and the book-a-visit functions	Yes, obviously only for the line-up	Time Estimation
R16	In case of deletions, the time could need to refresh its value	No, the estimation time remains as-is. Moreover, the refresh could cause problems.	---
R17	The estimation time must be based on the historical data	Yes	Time Estimation
R21	Fallback options must include all the product functions	Yes, managers and employees can make new reservations. We assume that those reservations	Queue Update

		are to be made	
R22	When it comes the turn of the lined-up customer, he must be able to get into the grocery store	Yes	Queue Update
R23	The hardware must support the entrance of the customer	Yes, in the sense that we provided functions that an eventual hardware system can invoke to register the entrance of a customer	Queue Update
R24	Registration must include name, username, email and password	Yes	Registration Module
R25	Registration must be possible from all the communication technologies supported	Yes	Registration Module
R26	Access must be possible providing email, or username, and password	Yes	Login Module
R27	Managers must be able to add a new grocery in the list	Yes	Grocery Handler
R28	Managers must be able to update informations about the store	Yes	Grocery Handler
R29	Managers must be able to register themselves to the system	Yes	Registration Module
R29	Managers must be able to register themselves to the system	Yes	Login Module
R30	Managers must be able to see data mined by the application	Yes	Monitor Module
R31	Managers must be able to add to the line a reservation	Yes	Queue Update
R32	Managers must be able to edit a reservation in the line	No, there is no need to edit a reservation when availability is ensured. It is equivalent to close an old	---

		reservation and open a new one.	
R33	Managers must be able to delete a reservation in the line	Yes	Reservation Handler
R34	Managers must be able to add employees	Yes	Employees Module
R35	Managers must be able to delete employees	Yes	Employees Module
R36	Employees must be able to log with the credentials provided by the manager	Yes	Login Module
R37	Employees must be able to manage both line-up and book-a-visit reservations such as managers	Yes, only for line-up	Reservation Handler

The missing requirements' number were related to not-implemented goals.

2.1. Further assumptions

In order to provide a prototype of the application we made some further assumptions partly yet explained in the rest of the section

1	The process that would require a check when a new grocery is added to verify the coherence of the data inserted has been skipped.
2	We have considered the id of the reservation as the QRcode. Indeed, since we do not have the possibility to have the hardware descripted in the RASD to read the QR code, we had to find something that mocked the feature of unicity.
3	Regarding the entrance and the exit of the customer, we used a system (that could be even real) so that an admin or an employee (employed for the right grocery) could set the state of a reservation as ENTERED or CLOSED (thus, entered than got outside the store). This is because, again, we do not have available the hardware technology that could allow the user to get into the store or to register their exit.
4	We modeled the Queue as the object that tracks all the reservations made for a certain grocery. Slightly different from what explained during the algorithms' paragraph on the DD

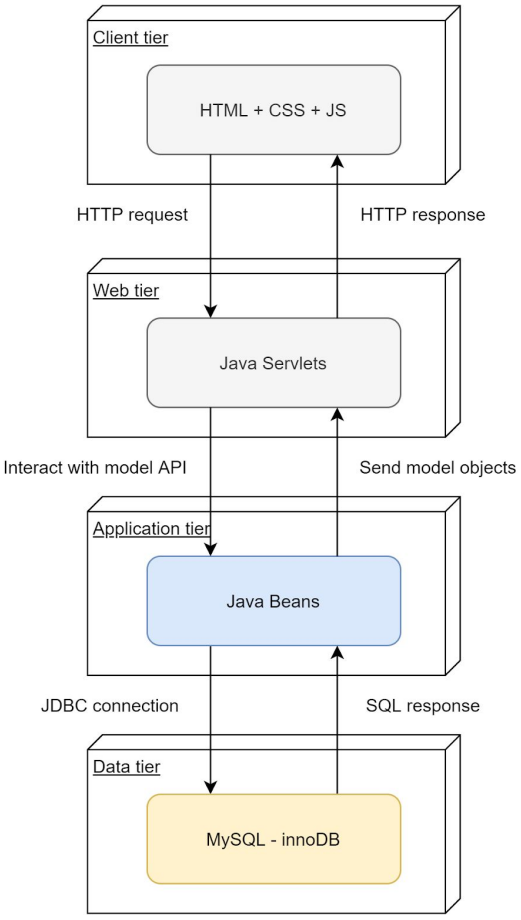
	in which the queue is only the actual queue to get into the store. Indeed, the implemented Queue contains even information about the past reservations made.

3. ADOPTED DEVELOPMENT FRAMEWORKS

This section is about the frameworks adopted during the implementation of the project. As already suggested during the DD's sections Deployment View [2.C] and Selected Architectural Styles and Patterns [2.F] the selected framework is that typical of a Java Enterprise Edition. We have done this choice for many reasons, partially yet exposed during the DD. The client-side scripting relies on HTML markup language, CSS styling and JS to add dynamism to the experience. We relied to the models already explained and designed in the DD such as:

- ER model to use it as a basis for our business data;
- Component interfaces' class diagrams to give a structure to the server-side implementation;
- IFML model in order to give a structure to the user experience in the web and client-side scripting.

To better appreciate the architecture and adopted frameworks used here we provide a very high-level diagram



This is a very high level diagram that gives an overall insight to the system development.
The logical separation between the components ensures a grade of scalability of the system, performance, and speed of development.

- scalability: any node can scale-up or downsize when needed. For example, in the JEE framework this is done automatically by the containers providers. For example the servlet container, in our case, Apache TomEE, can decide to balance the load to a certain servlet given the number of HTTP requests that arrive from the client ;
- performance: every node is developed on a specific software. This ensures that every optimization is ad-hoc to the specific node interested;
- speed of development: this is because implementing independent tiers can mean to implement those components in parallel. In our case, a component played as back-end developer (application and data tiers) while the other as front-end developer (web and client tiers). This improves the efficiency and cuts the time needed.

The communication protocols used are HTTP for the communication between the web tier and the client tier. For the prototype, we developed the web, application, and data tier on the same physical node. Those are in any case logically divided and even the source code is distinguished among each other.

3.1. Client Tier

The client tier, aimed at rendering the UI to the client, is structured over HTML, the style described through CSS and the client-side scripting is made through JS. This is made to make the UX as much user-friendly as possible thanks to CSS styling and JS asynchronous calls and checks. HTML is the markup language thanks to the servlets being able to publish pages to the clients. We used Thymeleaf in order to dynamically create those pages, while providing useful tools to support navigation of Java collections. Indeed, thanks to Thymeleaf and its engine, it is possible to specify POJOs or collections of them as variables in the context. It will be Thymeleaf that will do the translation between the POJO and the JSP template, relying on HTML.

3.2. Web Tier

The web tier uses the servlets and here the web server provider plays a relevant role. Again, the web server provider, or Apache TomEE, is responsible for all the concerns of the servlets which are basically Java classes. Starting from the lifecycle to the thread-safeness, the web server is the main actor of the web communication. Adapting this approach can make the development deeply dependent on a certain platform, but doing so we do not have to think about anything but functional requirements of the system.

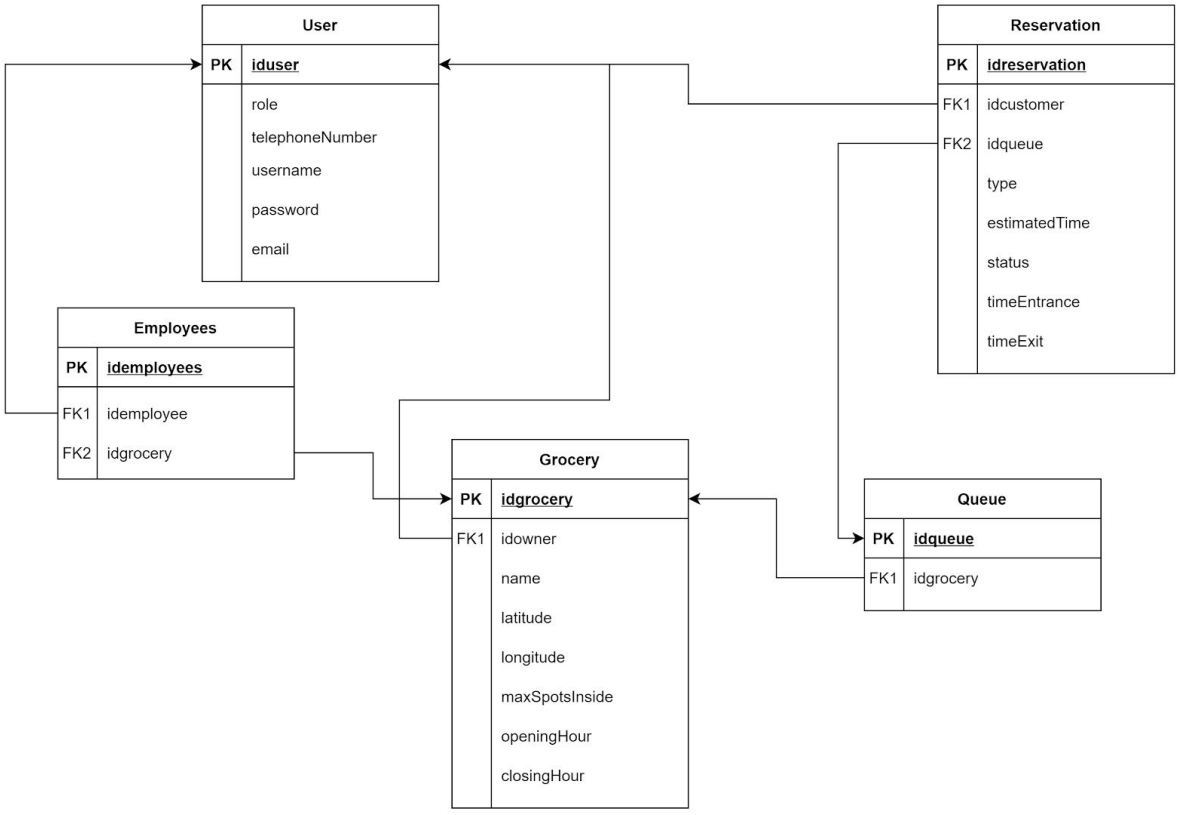
3.3. Application Tier

The application tier has been implemented with Java and JPA. Indeed, JPA supports the ORM (object-relational mapping) and helps with the alignment between the entities stored into the database and some enriched POJO in the Java program. This, again, makes the development totally dependent on a certain platform while taking away the responsibility to the programmer to manage the connection with the database explicitly with JDBC exposing to possible threats that are even

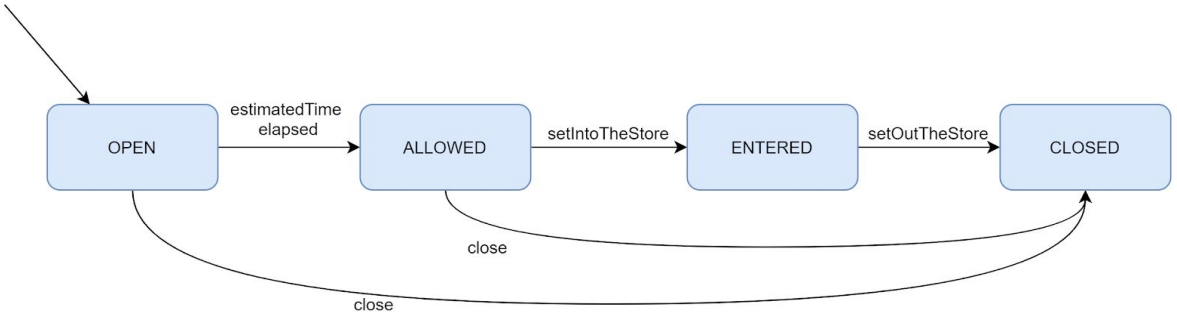
more sensible when talking about business data. JPA does all this work implicitly, or “under-the-hood”, while controlling the ACID properties with the only drawback to using some annotation in the Java code . Moreover, this approach can make the implementation independent from the DB since the annotations can be true for more than one version of DB vendor.

3.4. Data Tier

Thanks to JPA, we did not care much about the management of data storage. We used MySQL with InnoDB as an engine. Here we provide a logic model of the business data exploited.



In order to be clearer about the capabilities of the Reservation object here we describe the different states that can have a reservation, described into the `enum ReservationStates`, in its lifecycle.



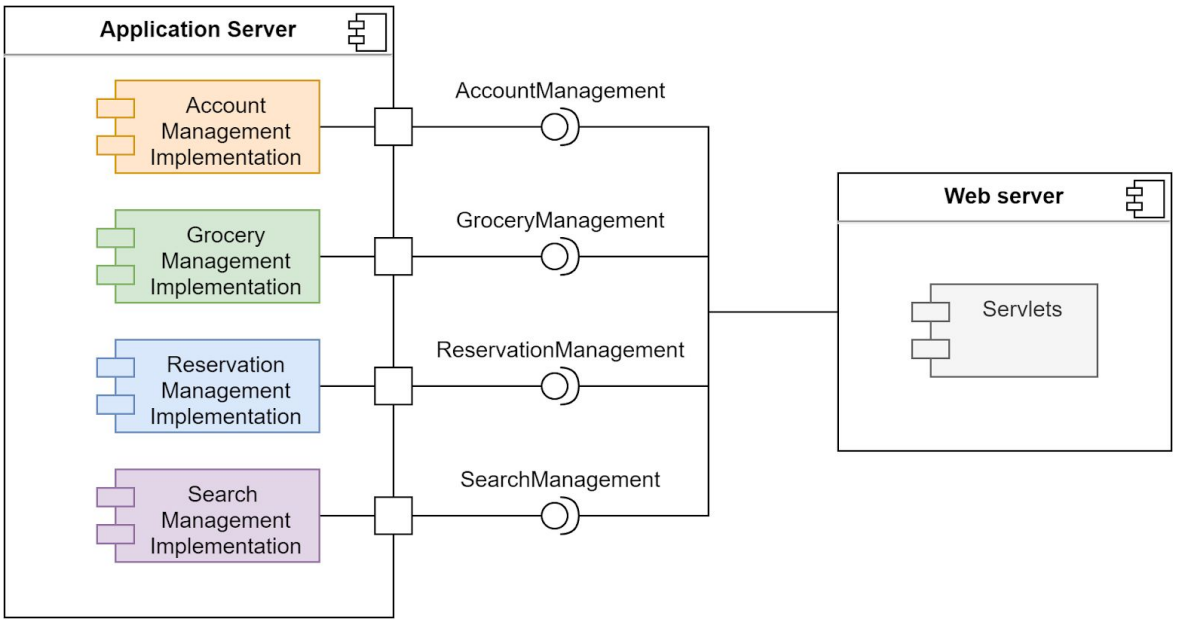
For more details about the estimatedTime, we remind that it was yet explained especially in the DD. We recommend to see even the dedicated part in the algorithms' section.

4. STRUCTURE OF THE CODE

Here we will expose the structure of our code. Since the development has been done independently on the front-end and back-end, we will expose these two separately. You can find attached to the source code the javadoc that describes the API. The code is in the repository under the SourceCode directory. While the front-end code is in the `SourceCode/WebTier` folder, the back-end is in `SourceCode/BusinessLogicTier` folder. Each folder contains a different project: obviously the web tier code uses the business logic one.

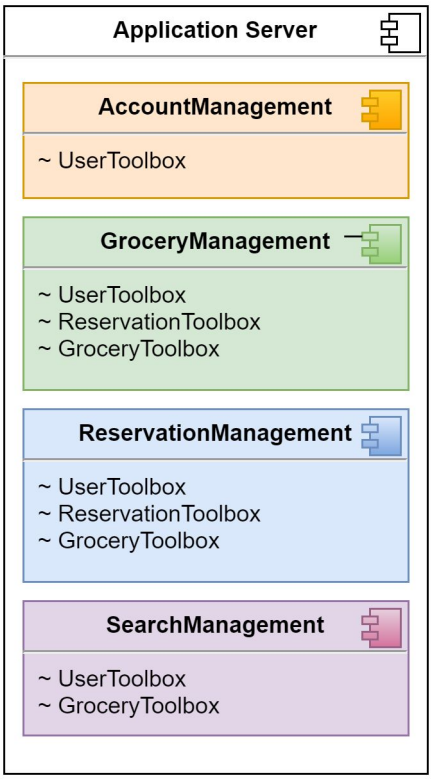
4.1. Back-end code

The code at server side is based on that has been explained in the Component Interfaces section of the DD [2.E]. The entities are under the "model" package. The business logic is implemented under the "services" package. Here there is a high-level representation of it.



Each module represented inside the application server is actually a package in the code under the "`services`" package providing both the interface of the sub-module and the implementation of it. Each sub-module has been already defined in the Component interfaces section of the DD[2.E]. Each interface exposed above is actually an interface under the "`macrocomponents`" package. Here, actually, these interfaces are stateless EJB that contain an injected entity manager instance and all the tools needed. In this context, tools, or toolboxes, are DAOs specific to a certain entity mapped to the database responsible to

centralize the invocations to the entity manager by means of `persist()`, `remove()`, `find()` calls and creation of named queries to do to the database through JTA. They simply forward the requests of the services to the entity manager. Actually, there are only 3 toolboxes: the ones referred to User, Reservation and Grocery entities.



Here, it is visible how these components are distributed among the `macrocomponents` classes.

4.2. Front-end code

The front-end code is divided between web and client tiers. The web code is essentially formed by Servlets: Java classes that implement the methods doGet and doPost that trigger when a HTTP get or a HTTP post is done to the server. HTML pages are created by the servlet and sent as HTTP responses to the client. Here, as already explained, Thymeleaf does its work and generates dynamic JSPs. Client-side, CSS styles the pages while JS, used for the map, is executed when needed.

5. TESTING SECTION

Depending on the technology behind, different tests have been made.

5.1. Back-end testing

Back-end testing has been performed with JUnit. The test plan has been performed according to the one specified in the DD.

- Unit tests: unit tests were performed before the integration tests. Here, the invocations of methods outside the tested class have been mocked by ad-hoc mock classes inside the test class.
- Integration tests: here, the classes involved by the test were actually only the toolboxes classes, tested together with the database. There was the need to seed the databases with some mock-data for then drop it to leave the environment clean. To do so, it has been implemented in 3 classes under the "`test/resources`" package. Finally, there was the need to create an EJB environment outside the Apache TomEE environment. For this reason, it has been used an Application-Managed persistence context rather than a transactional-scoped one, specifying the JDBC data inside the `META-INF/persistence.xml` file.

Here we provide some coverage statistics mined by Eclipse with JUnit during the tests:

- Types coverage:

Element	Coverage	Covered Types	Missed Types	Total Types
exceptions	0.0 %	0	1	1
src.main.java.exceptions	100.0 %	1	0	1
src.main.java.model	100.0 %	6	0	6
src.main.java.services.accountManagement.implementation	100.0 %	2	0	2
src.main.java.services.accountManagement.interfaces	100.0 %	2	0	2
src.main.java.services.groceryManagement.implementation	100.0 %	3	0	3
src.main.java.services.groceryManagement.interfaces	100.0 %	3	0	3
src.main.java.services.macrocomponents	100.0 %	4	0	4
src.main.java.services.reservationManagement.implementation	100.0 %	6	0	6
src.main.java.services.reservationManagement.interfaces	100.0 %	5	0	5
src.main.java.services.searchManagement.implementation	100.0 %	1	0	1
src.main.java.services.searchManagement.interfaces	100.0 %	1	0	1
src.main.java.services.tools	100.0 %	3	0	3

- Instructions coverage:

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
exceptions	0.0 %	0	4	4
src.main.java.exceptions	100.0 %	4	0	4
src.main.java.model	81.8 %	404	90	494
src.main.java.services.accountManagement.implementation	92.7 %	177	14	191
src.main.java.services.accountManagement.interfaces	42.9 %	6	8	14
src.main.java.services.groceryManagement.implementation	94.6 %	487	28	515
src.main.java.services.groceryManagement.interfaces	42.9 %	9	12	21
src.main.java.services.macrocomponents	100.0 %	12	0	12
src.main.java.services.reservationManagement.implementation	87.0 %	793	119	912
src.main.java.services.reservationManagement.interfaces	54.3 %	19	16	35
src.main.java.services.searchManagement.implementation	90.1 %	136	15	151
src.main.java.services.searchManagement.interfaces	42.9 %	3	4	7
src.main.java.services.tools	89.4 %	194	23	217

- Branches coverage:

Element	Coverage	Covered Branches	Missed Branches	Total Branches
exceptions		0	0	0
src.main.java.exceptions		0	0	0
src.main.java.model	75.0 %	12	4	16
src.main.java.services.accountManagement.implementation	81.2 %	39	9	48
src.main.java.services.accountManagement.interfaces		0	0	0
src.main.java.services.groceryManagement.implementation	85.4 %	76	13	89
src.main.java.services.groceryManagement.interfaces		0	0	0
src.main.java.services.macrocomponents		0	0	0
src.main.java.services.reservationManagement.implementation	85.3 %	58	10	68
src.main.java.services.reservationManagement.interfaces		0	0	0
src.main.java.services.searchManagement.implementation	76.9 %	20	6	26
src.main.java.services.searchManagement.interfaces		0	0	0
src.main.java.services.tools		0	0	0

5.2. Front-end testing

Front-end testing has been performed with JUnit and Mockito.
The aim of these tests was to verify the correct behaviour of the logic behind the servlets.
All the test were structured as such:

1. Create the Mock classes for each controller to test and the respective EJB Mocks and HTTP objects mocks.
2. A @Before method which contains the construction of the mocked servlet, spied by Mockito to see the methods called inside of the object, and contains also a standard behaviour for some methods that have to be mocked, in particular the gets and post templates that are utility classes that exist to separate the controller logic to the Thymeleaf processing which is unnecessary to test.
3. Then we pass on to the test methods that have a precise inner structure:
    a. First we create any instance of entities we need for the test.
    b. Then we assign the correct behaviour to mocked objects with Mockito when(object.someMethod).thenReturn(someObject)

or thenThrows(someException), to emulate the behaviour of outside components.

c. Then we proceed to run the doGet or doPost method of the servlet.
d. Finally we verify that the correct methods have been called.

Moreover we can underline a division for each TestClass, tests are divided in categories:

e. NullParametersTest: check the behaviour of the servlet if null parameters are passed.
f. BadParametersTest: check the behaviour of the servlet if bad formatted parameters are passed.
g. ServletLogicTest: checks on special qualities that the servlet must have to work properly.
h. DatabaseErrorTest: check the behaviour of the servlet if some exception is thrown on the application server.
i. SuccessTest: check the behaviour of the servlet when everything goes as intended and the servlet process or redirect the response.

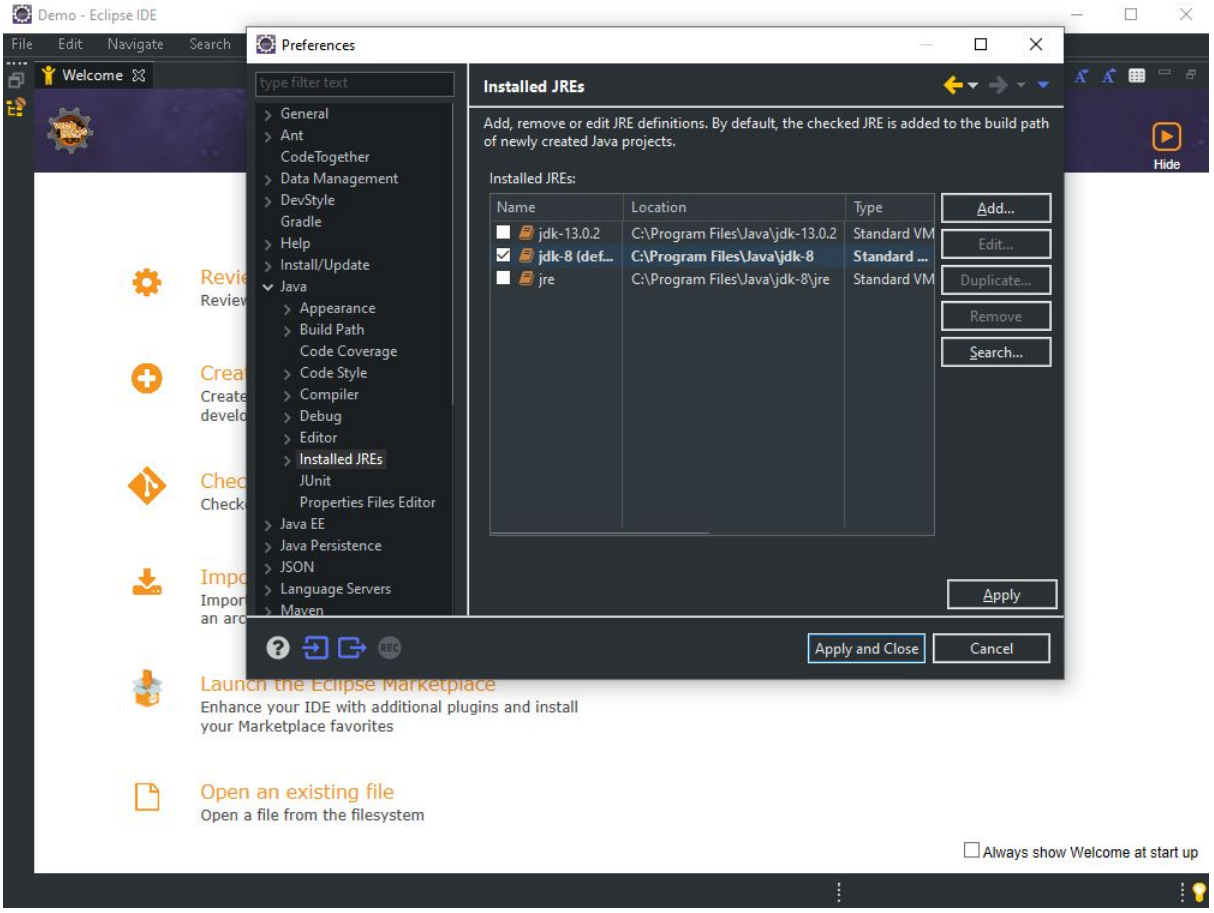
6. INSTALLATION INSTRUCTIONS

The artifact created is a zip file containing both source codes and a dump of the database prepared with some sample data. The following software is needed:

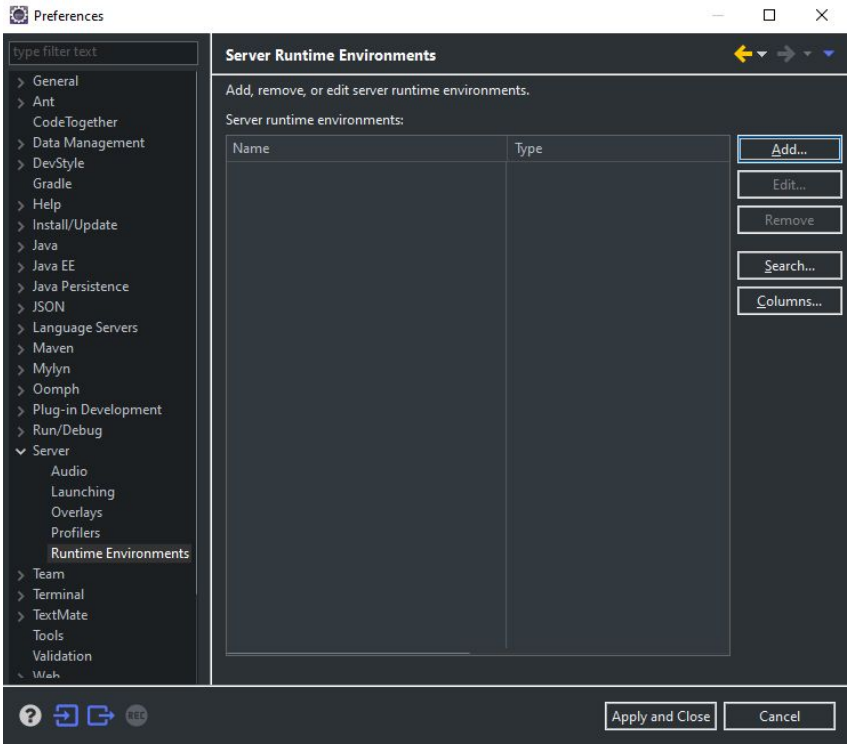
- an IDE, preferably Eclipse because the project was developed in that framework;
- a web server with JPA compatibility, we suggest Apache TomEE 8.0.3;
- a MySQL Server version 8.0.22 where to put the dump of the database. For simplified access to the database and its properties we recommend to use MySQL Workbench.

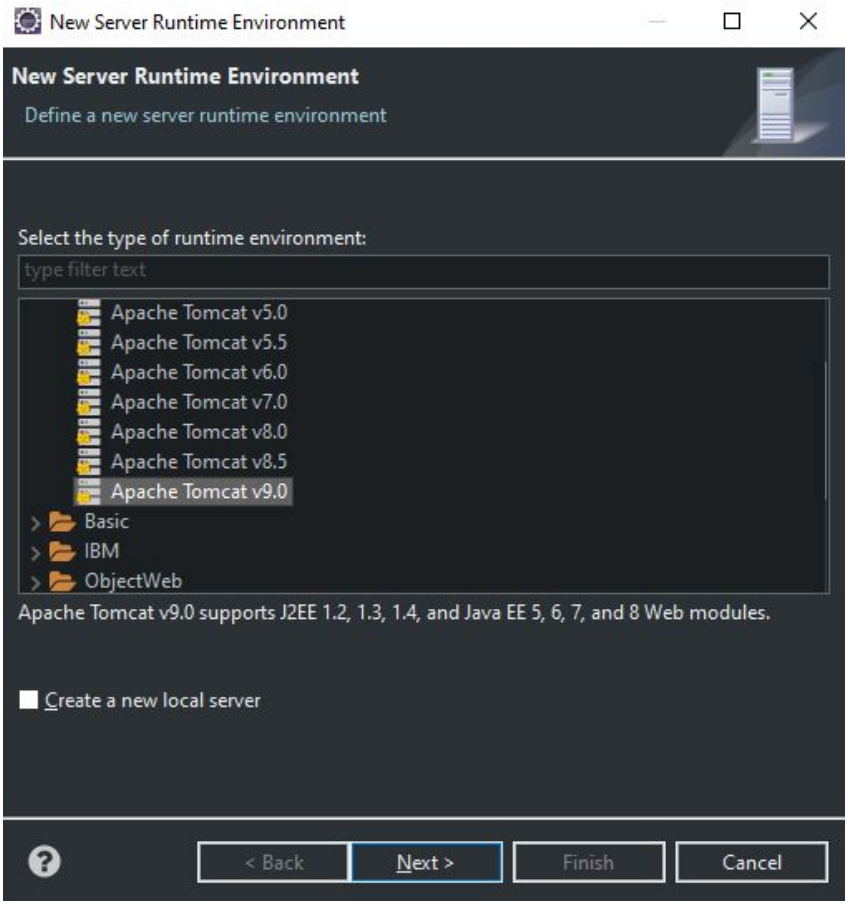
Here are the instructions to follow to install the project in Eclipse. There are many other ways to deploy this project. Since we used this environment we suggest this procedure but we added to the repository even a WAR file to deploy in any web server.

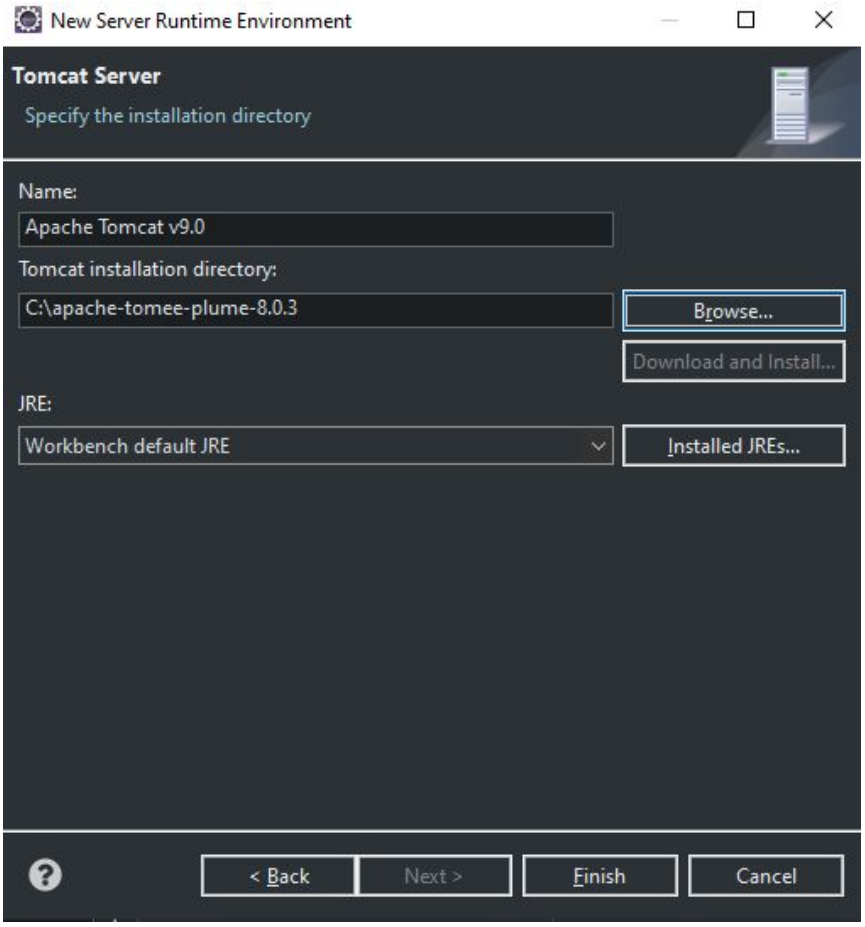
1. First of all download the zipped source code and the dump. The dump is already filled with some sample data, watch it to run a demo.
2. Install a version of the JDK, we recommend JDK 8 to avoid any problem.
3. Open your Eclipse, create a new workspace and set up the JDK you previously installed going to Window > Preferences >Java > Installed JRE and select it, then Apply and close.



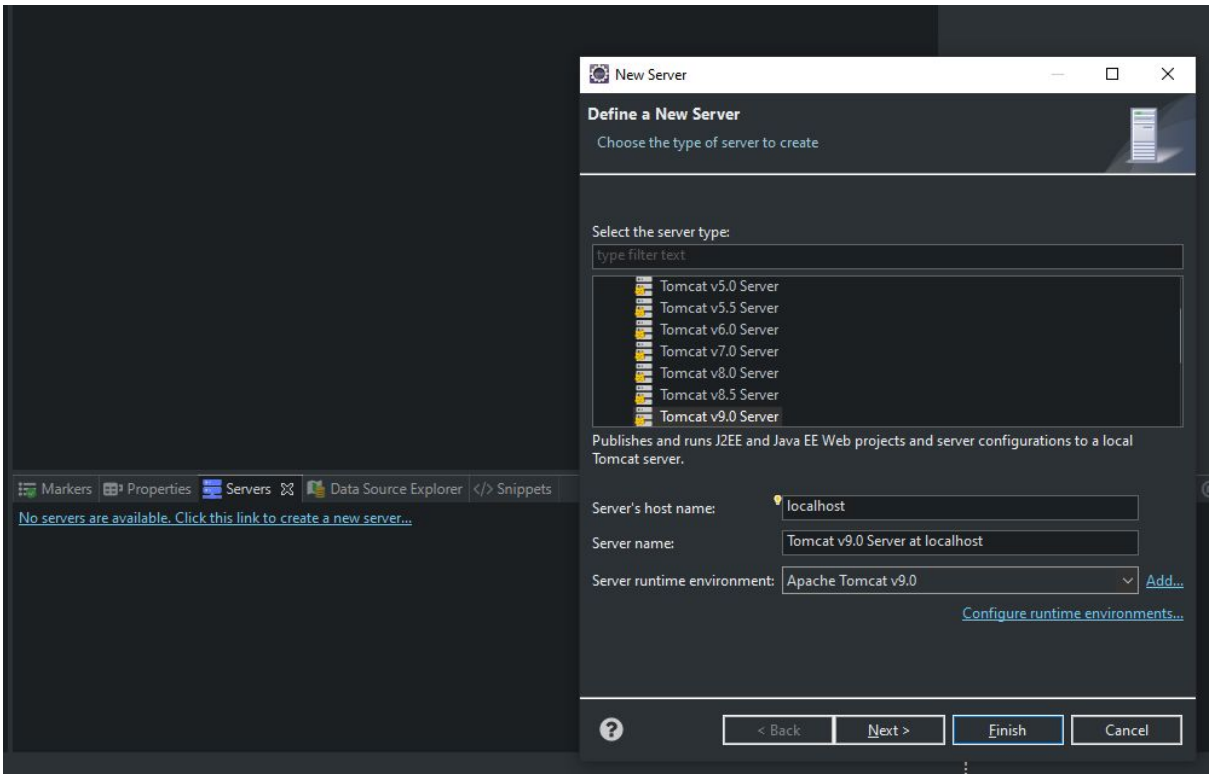
4. Then add your server (TomEE) to Eclipse going to Window > Preferences > Server > Runtime Environments and press add. Then select the version 9.0 of Apache Tomcat, click next and browse to the installation directory of your TomEE and click finish.


Preferences
type filter text
General
Ant
CodeTogether
Data Management
DevStyle
Gradle
Help
Install/Update
Java
Java EE
Java Persistence
JSON
Language Servers
Maven
Mylyn
Oomph
Plug-in Development
Run/Debug
Server
Audio
Launching
Overlays
Profilers
Runtime Environments
Team
Terminal
TextMate
Tools
Validation
Server Runtime Environments
Add, remove, or edit server runtime environments.
Server runtime environments:
Name
Type
Add...
Edit...
Remove
Search...
Columns...
Apply and Close
Cancel

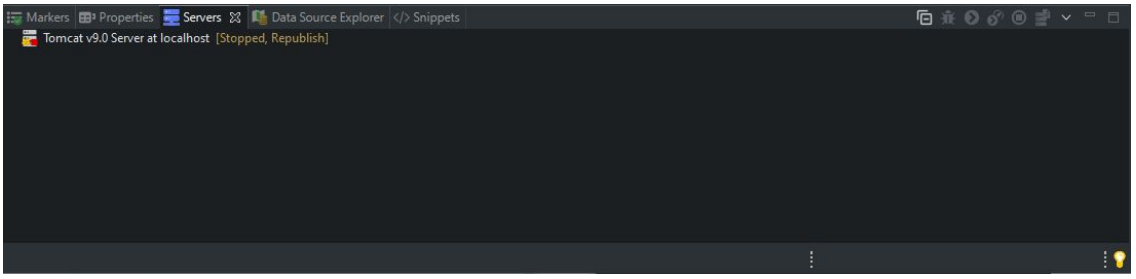

New Server Runtime Environment
New Server Runtime Environment
Define a new server runtime environment
Select the type of runtime environment:
type filter text
Apache Tomcat v5.0
Apache Tomcat v5.5
Apache Tomcat v6.0
Apache Tomcat v7.0
Apache Tomcat v8.0
Apache Tomcat v8.5
Apache Tomcat v9.0
Basic
IBM
ObjectWeb
Apache Tomcat v9.0 supports J2EE 1.2, 1.3, 1.4, and Java EE 5, 6, 7, and 8 Web modules.
Create a new local server
< Back
Next >
Finish
Cancel



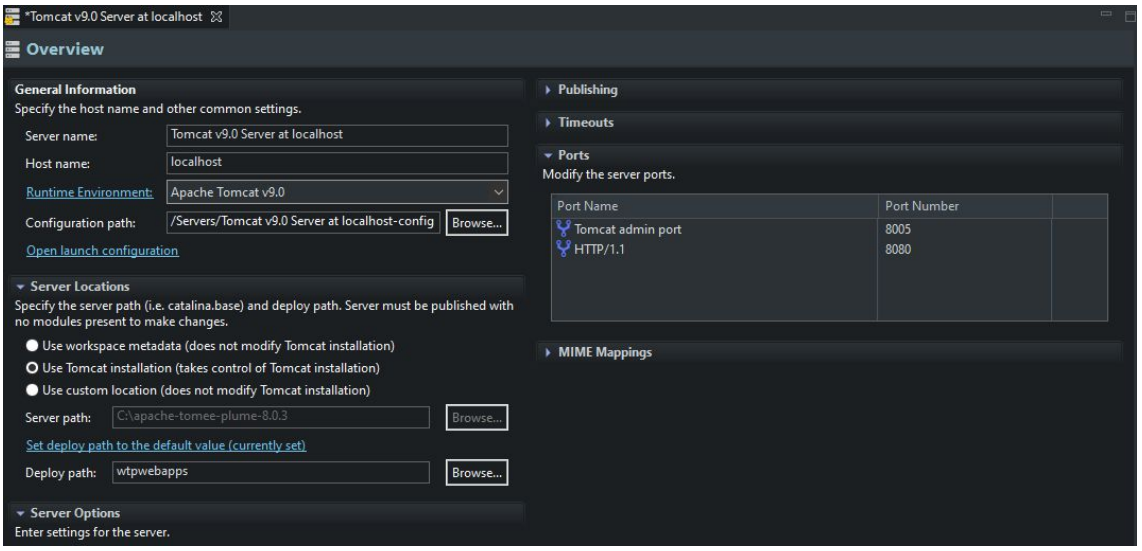
5. Go in the tab Servers of Eclipse (if not visible go to Window > Show view > Servers) and use the command to create a new server, selecting the runtime environment we just created.



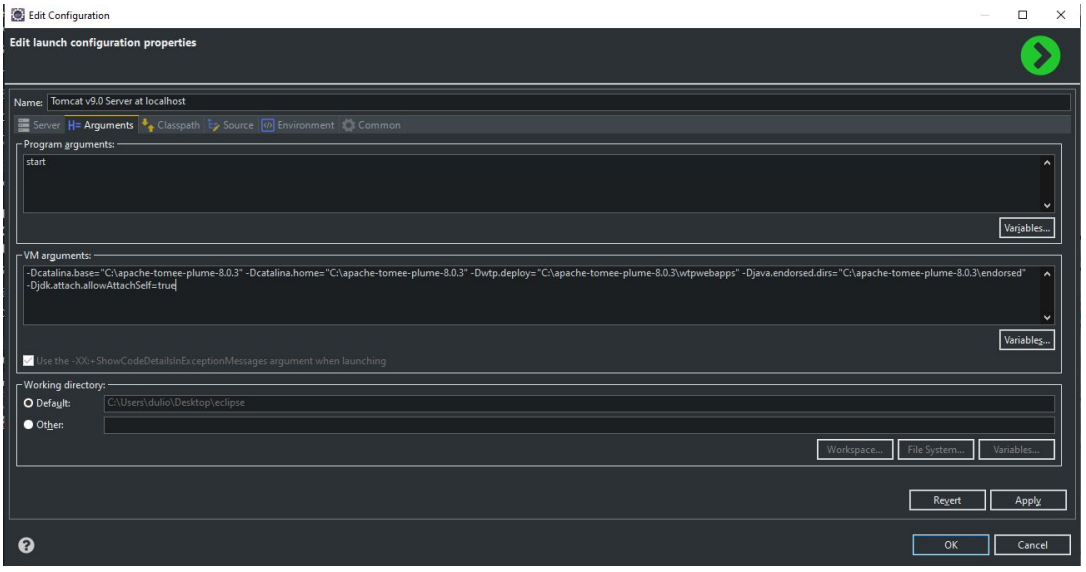
6. Now go in the servers tab and you should see the server, double click on it.



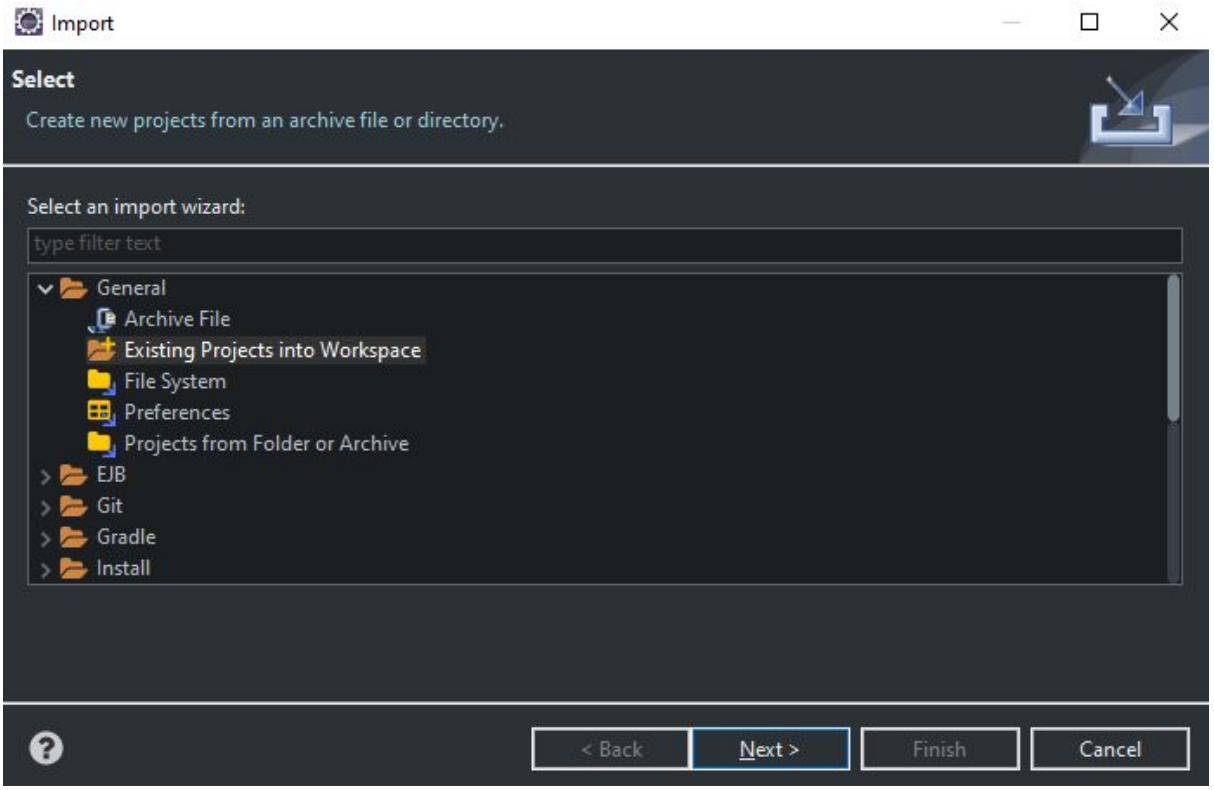
7. The server properties will open. Under the Server Locations section, select the option “Use Tomcat installation (takes control of Tomcat installation)”.



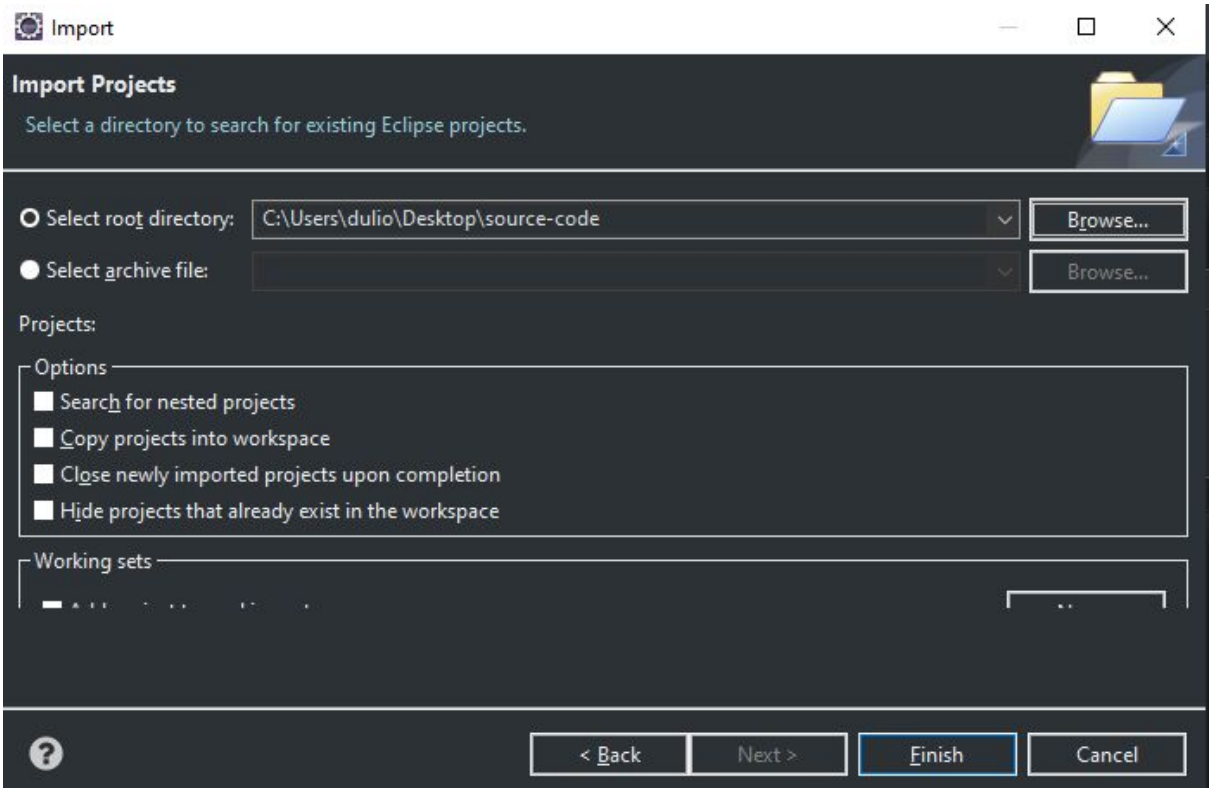
8. Now press on “Open launch configuration” and in the new window select the “Arguments” tab. Here you have to add “-Djdk.attach.allowAttachSelf=true” after all the VM arguments as shown in the figure below and press ok.



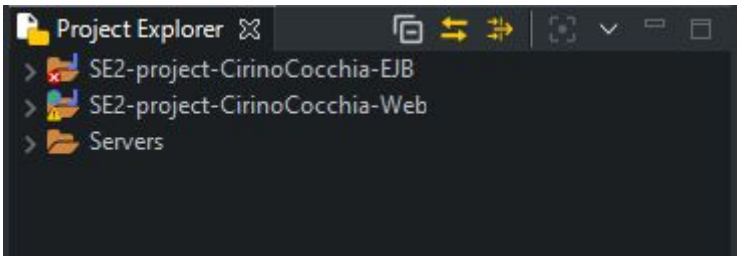
9. At this point you should load the dump of the database in your MySQL server.
10. Now go on File > Import > and select General > Existing Projects Into Workspace, click Next.



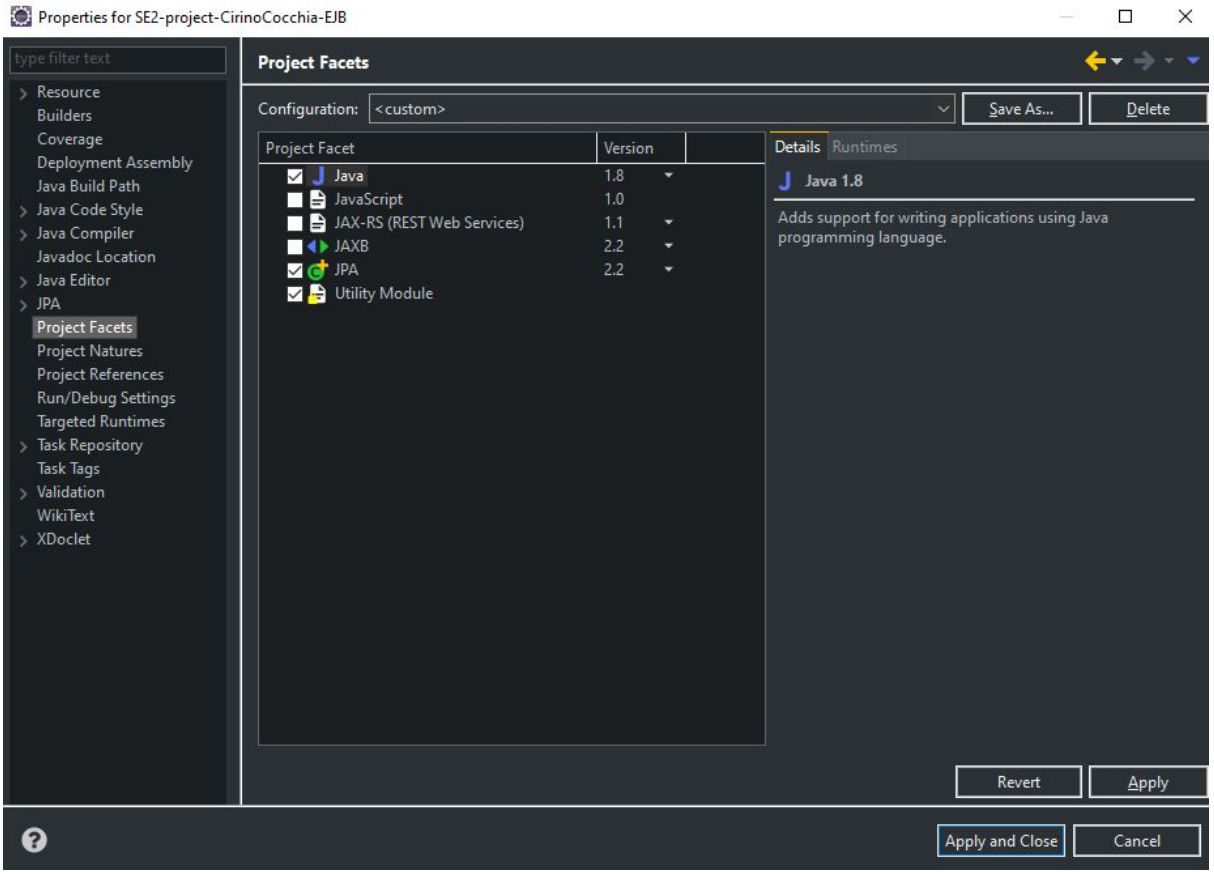
11. Select the unzipped file and click on Finish.



12. Now you should see the two projects in the Project Explorer tab. Right click on the web project and select properties. Navigate to Deployment assembly and verify that the EJB project is included. Else add it. Then click apply.



13. Now right click on the EJB project and go to Properties > Targeted Runtime and select your TomEE server you created before, then click apply.
14. Always in the properties section of the EJB project check that the Project Facets are versioned like in the picture below.



15. Then add the resource in the TomEE server instance in the file tomee.xml and modify the persistence.xml of the EJB part to point to these new resources shown below. (Be sure to have put the connector/j jar inside the {tomEE installation path}/lib folder, you can find it in the *WEB-INF/lib* folder).

RESOURCE TO PUT IN THE tomee.xml

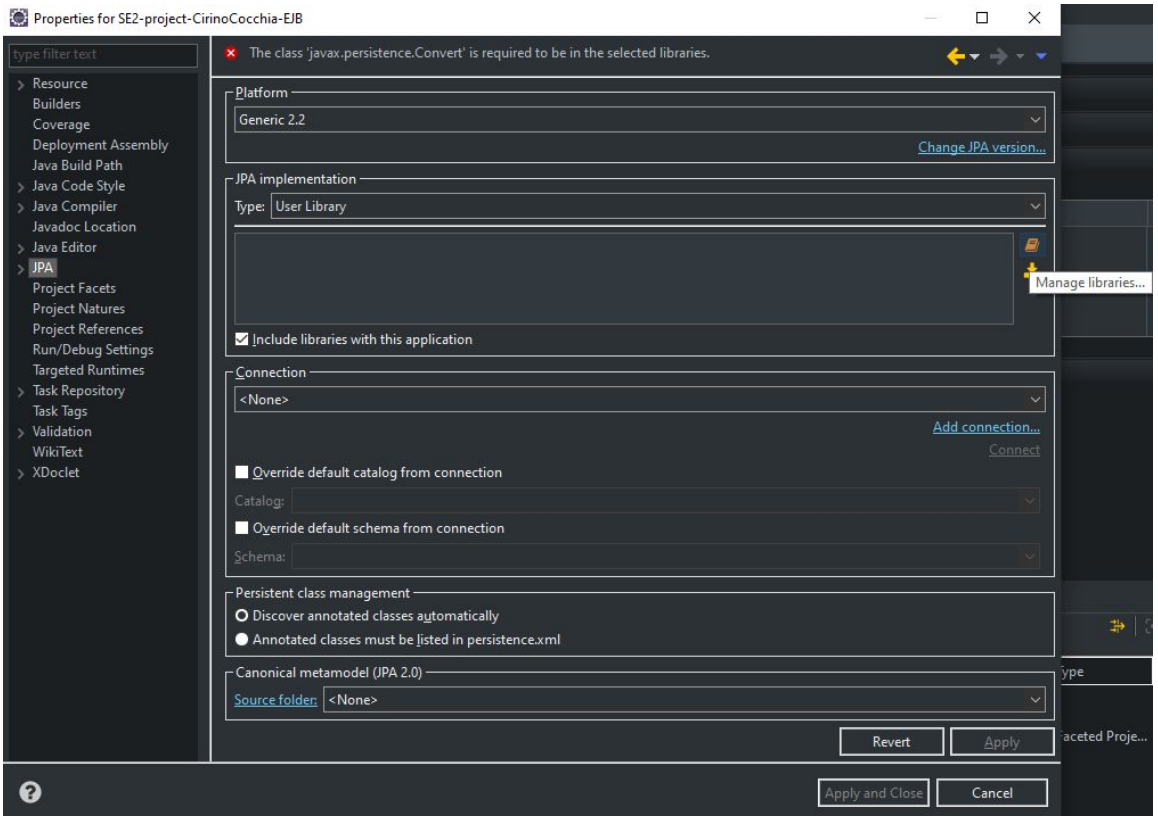
<?xml version="1.0" encoding="UTF-8"?>

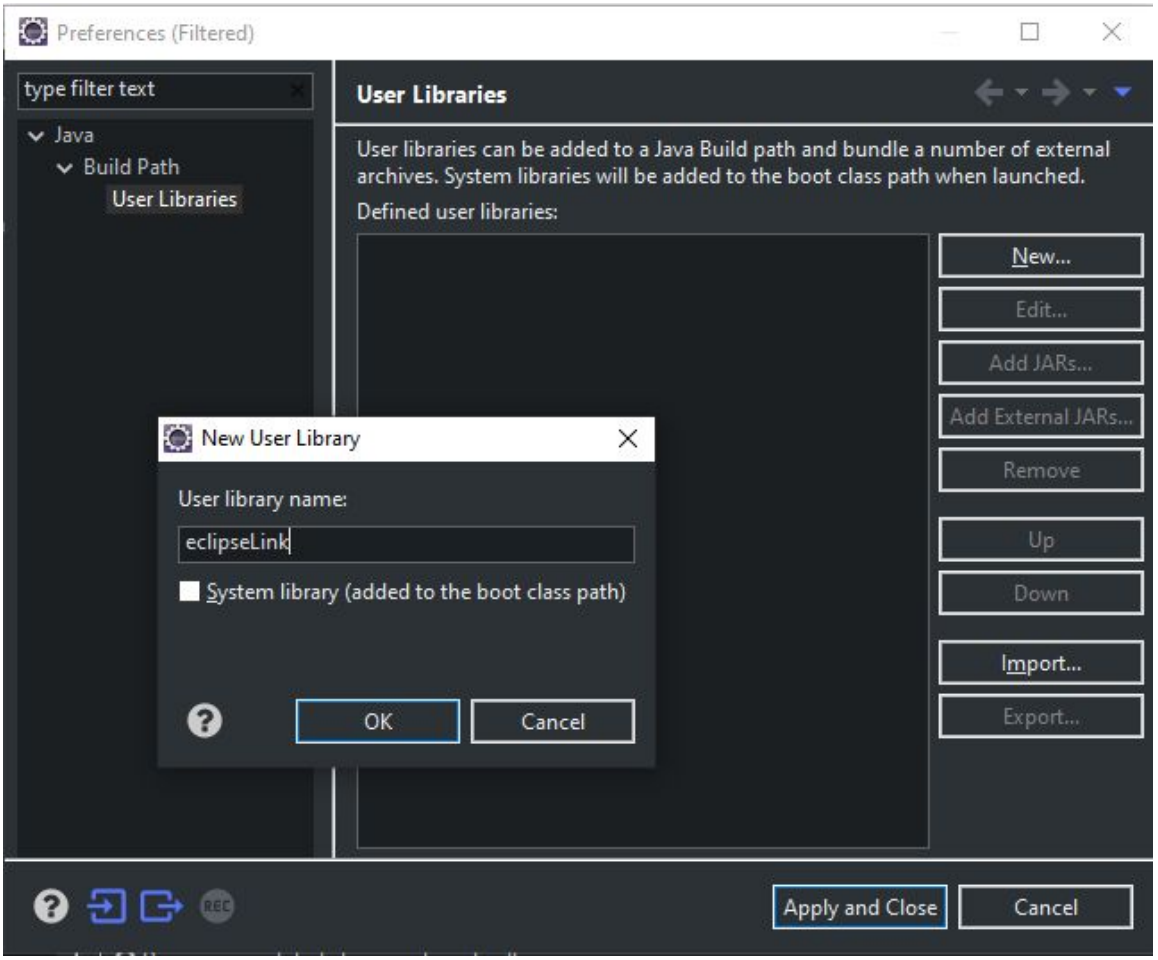
-<tomee>

<Resource type="DataSource" id="SE2-project-resource"> JdbcDriver com.mysql.cj.jdbc.Driver JdbcUrl jdbc:mysql://localhost:3306/db_project_se2 UserName yourUsername Password yourPassword</Resource>

</tomee>

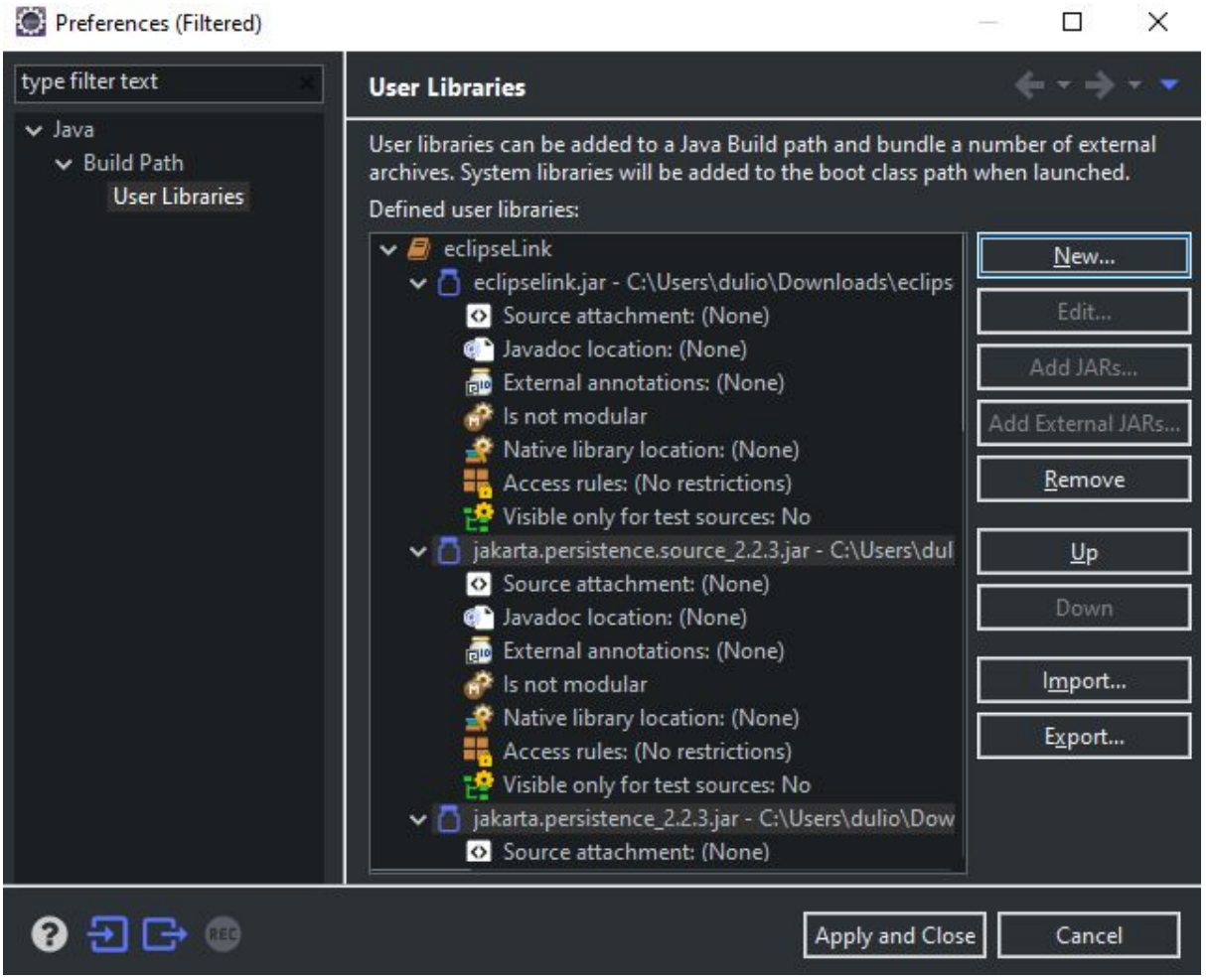
16. Then download EclipseLink 2.7.7 at this link, for any problem with this library contact us.
17. Go in the JPA section of Preferences and add a new User Library like in the pictures below.





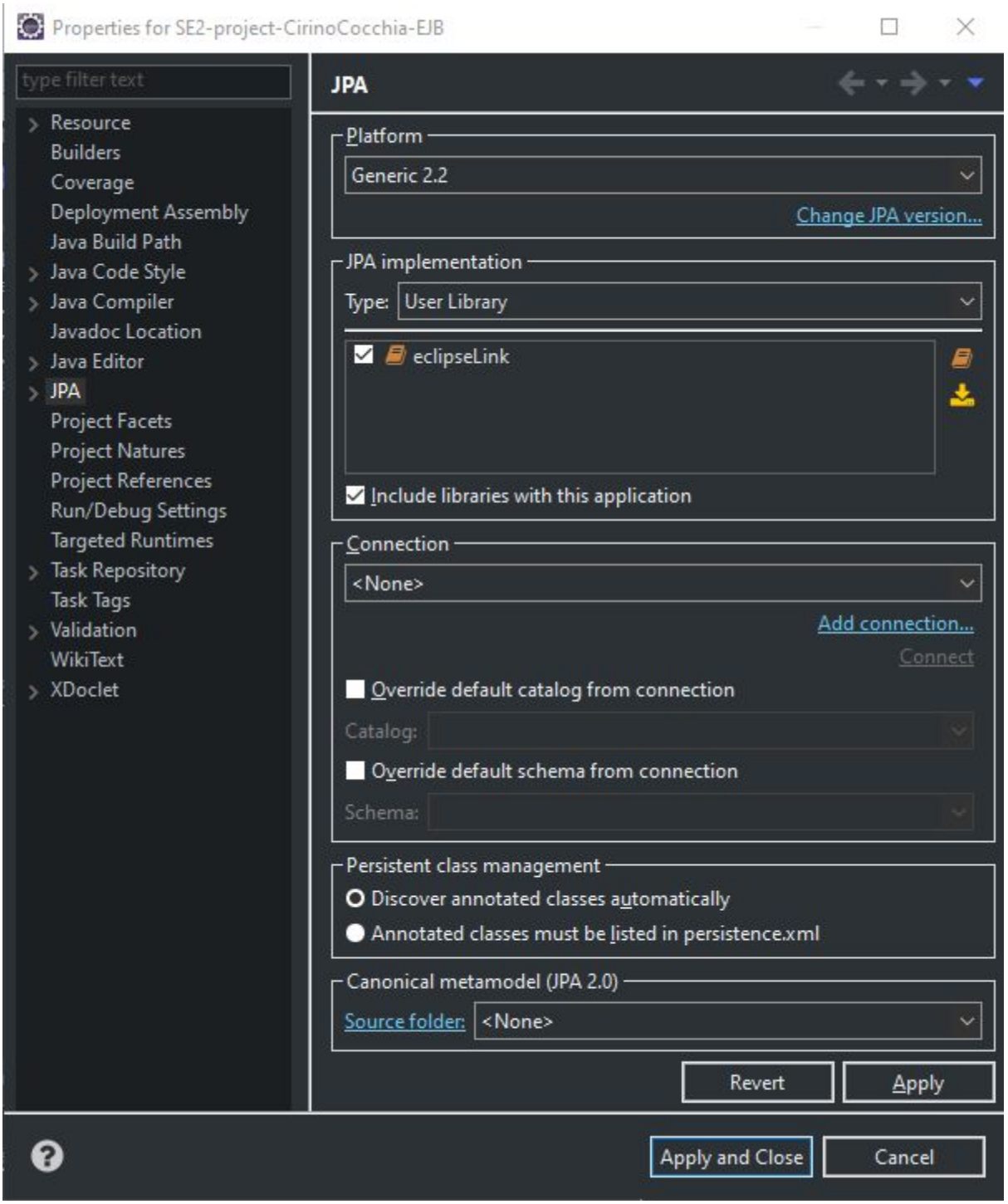
Click OK.

Now click on *Add External JARs* on the new library and add from *jlib: eclipselink.jar* and the ones in the *jpa* folder, these should be found in the EclipseLink 2.7.7 folder you previously downloaded.

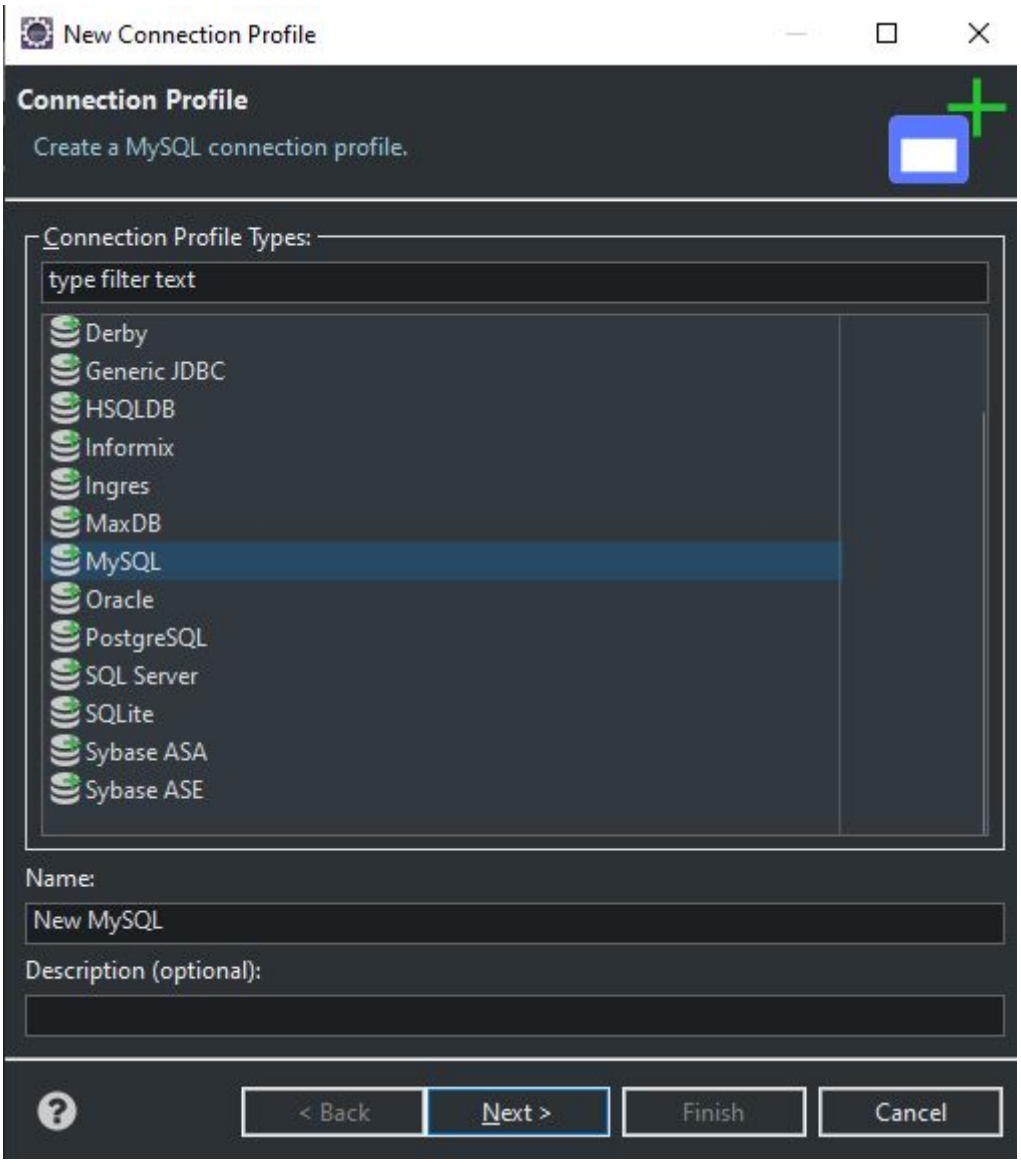


18. Now you have to create a connection to the MySQL database, be sure to download mysql-connector-java-5.1.49-bin.jar from the internet. Proceeds as in the steps below.

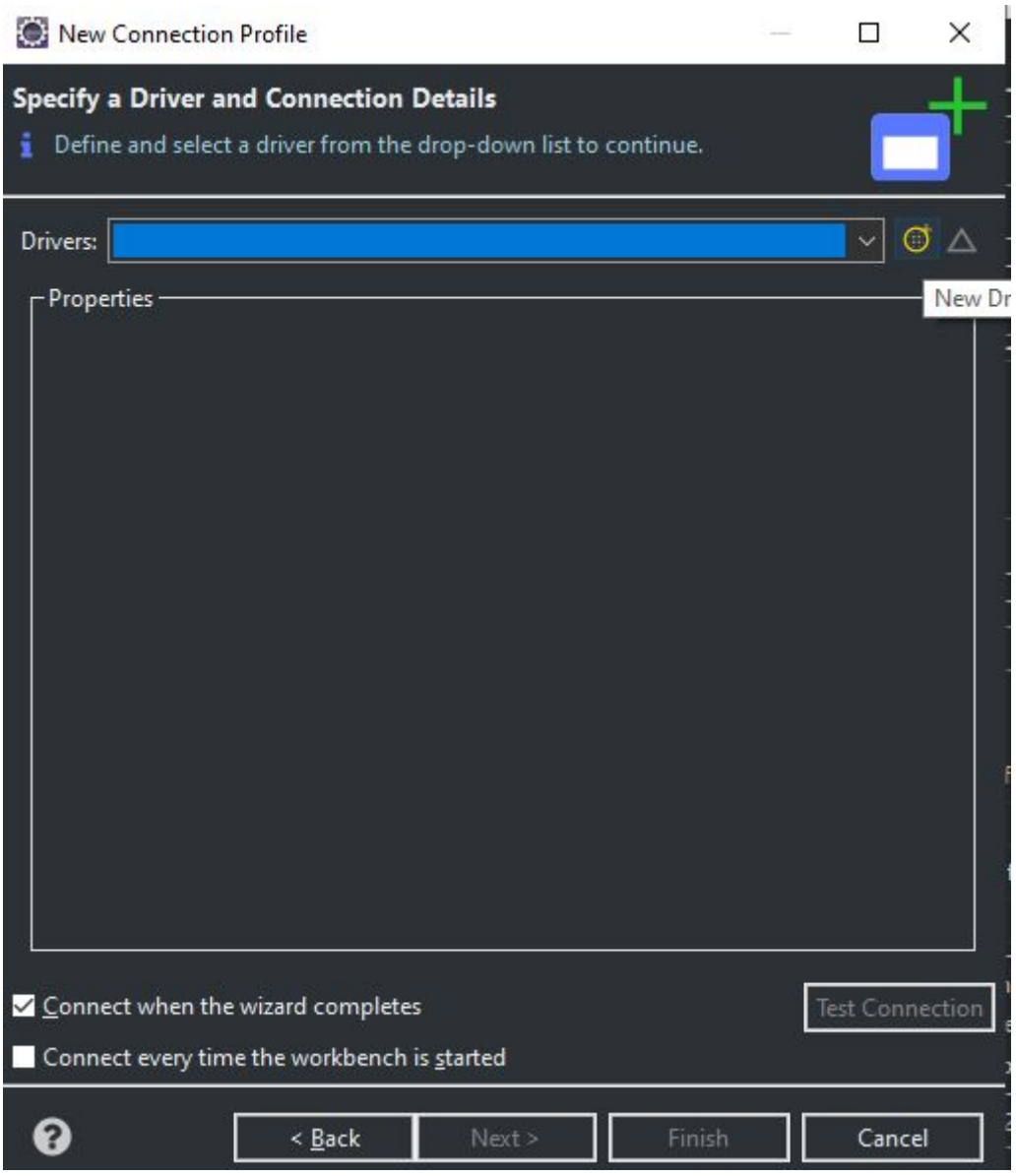
- Click on Add connection...



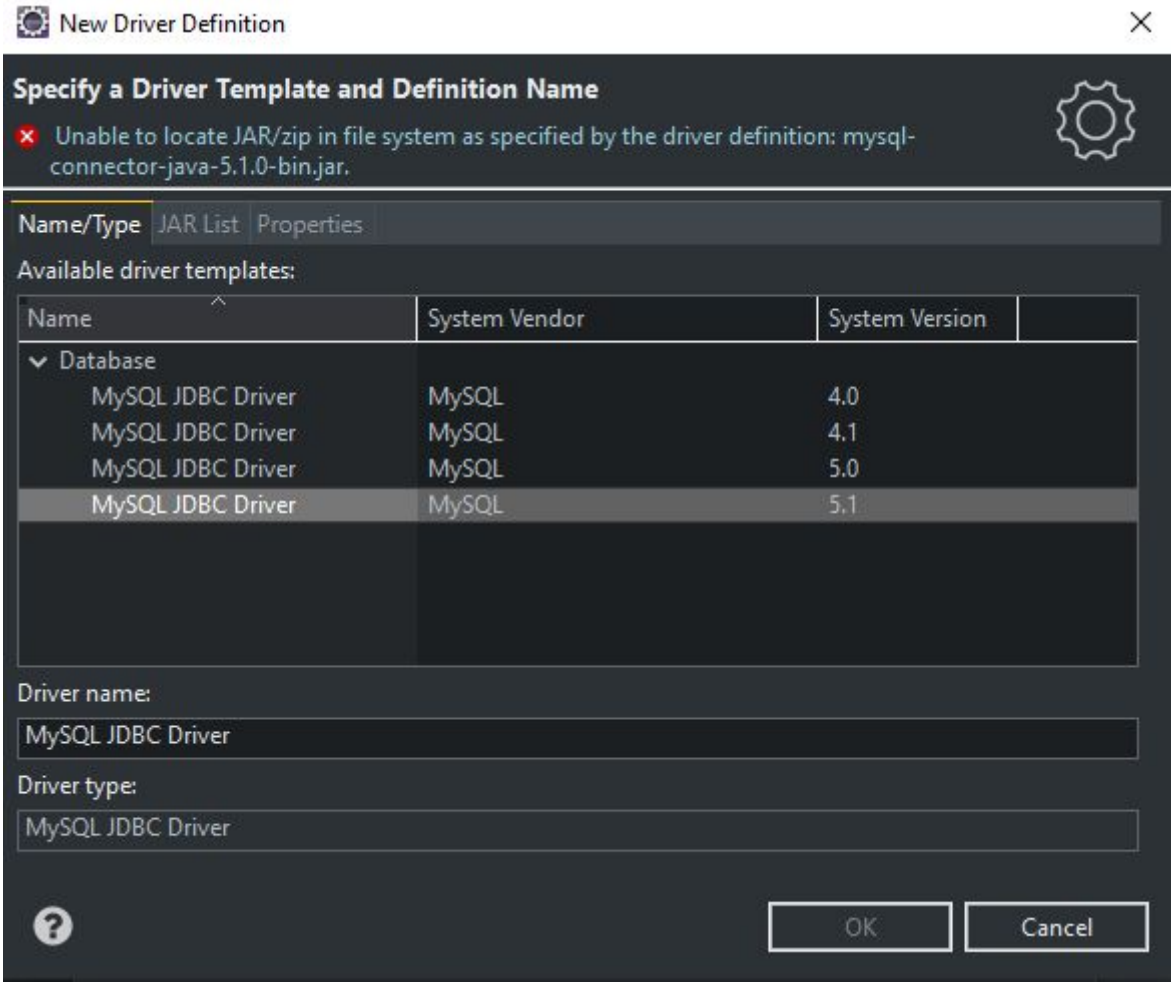
- Select MySQL as a connection profile and click Next



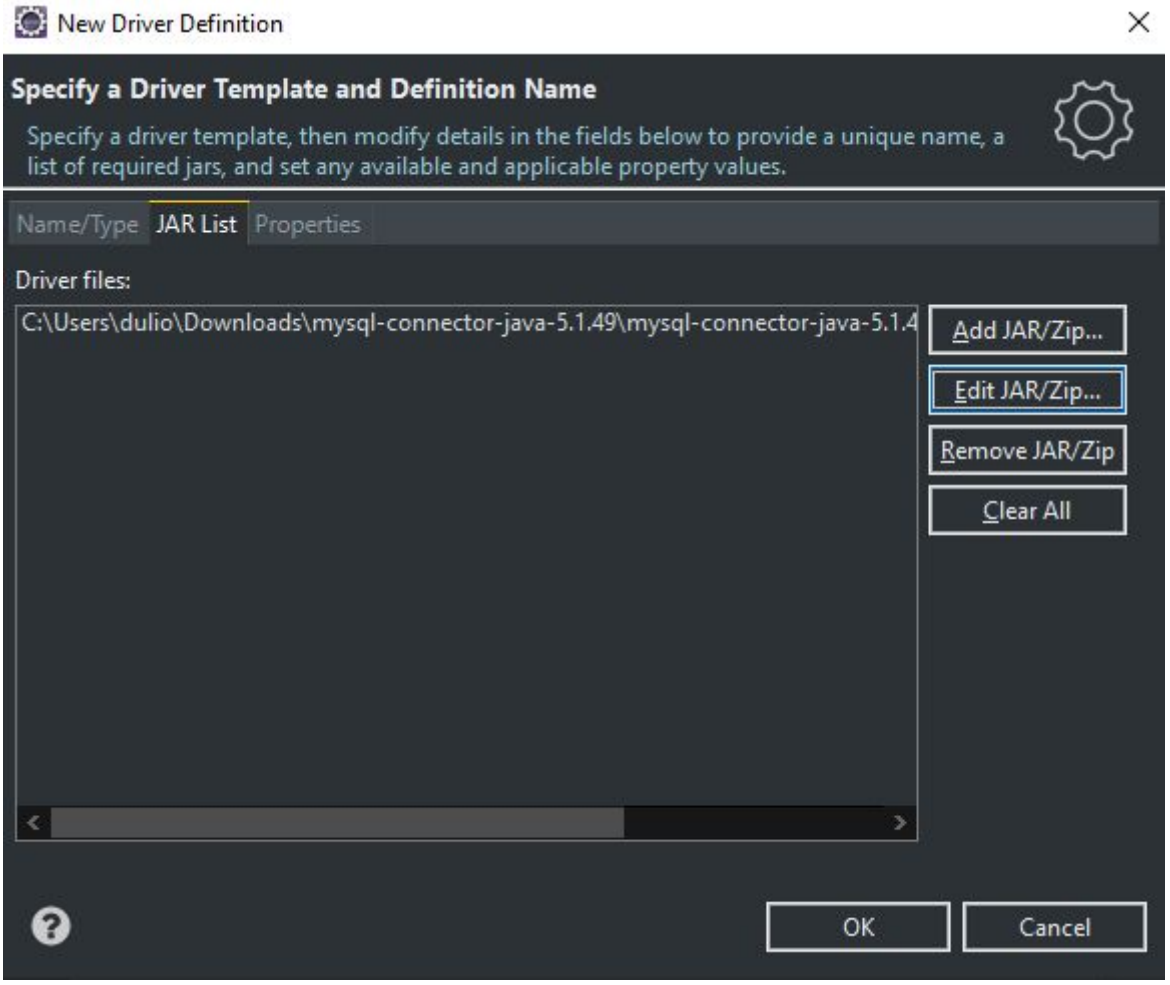
- Create a new driver clicking as shown in the picture



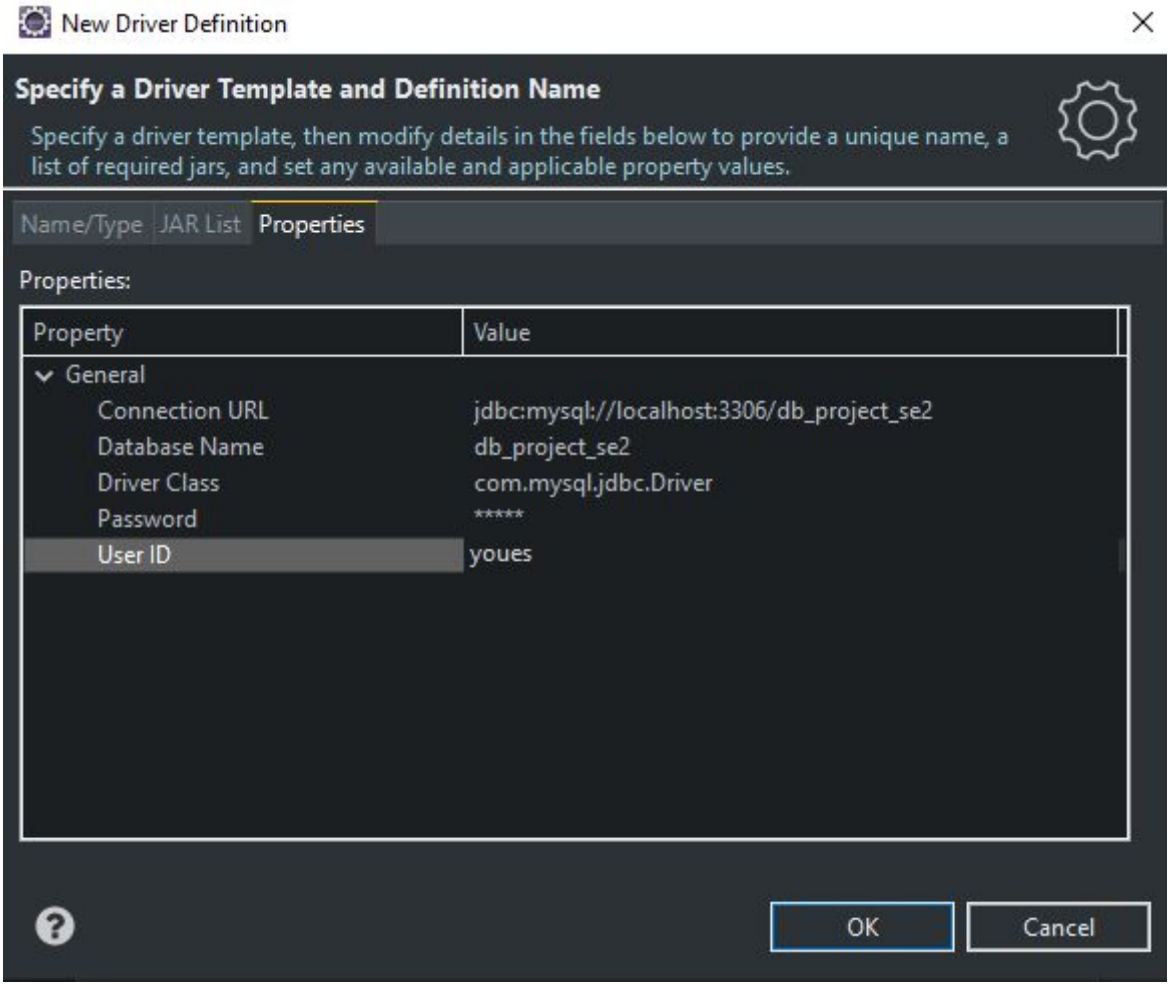
- Select MySQL JDBC Driver version 5.1



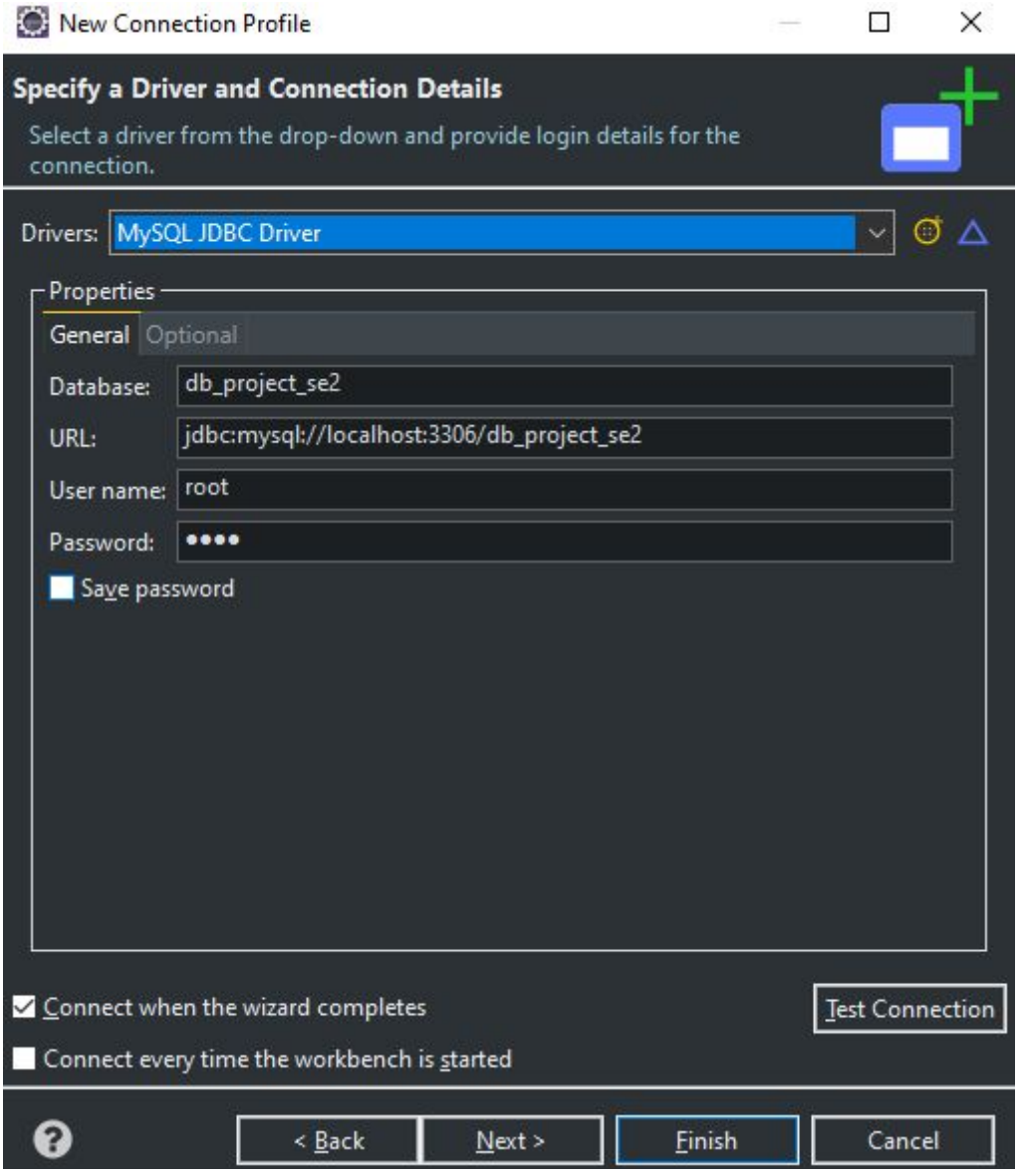
- In the JAR List tab, click on Add JAR (or Edit JAR if a not found one exists) and select the connector you previously downloaded.



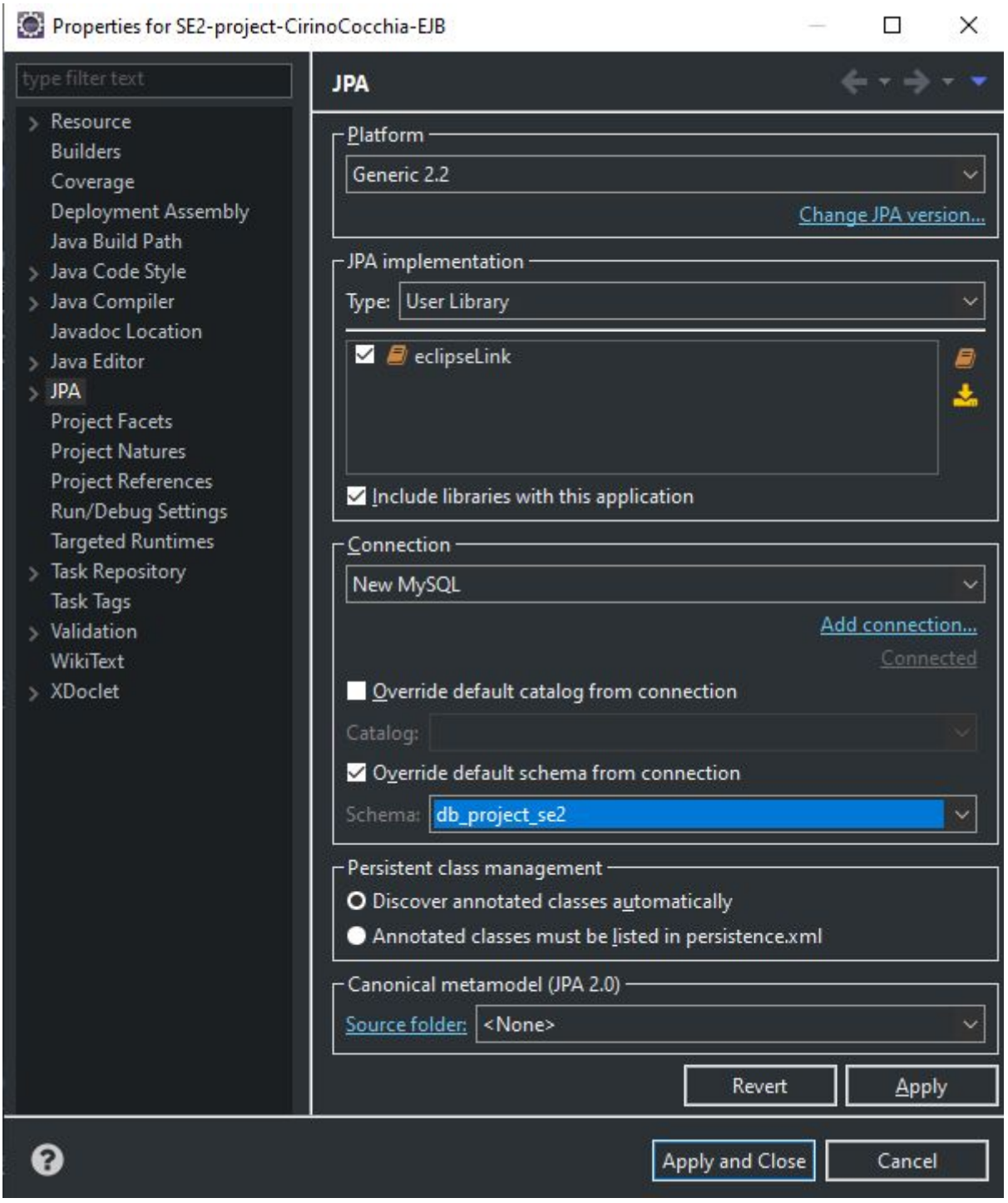
- In the properties tab fill the sections as in the picture below with your username and password to the MySQL Server.



- After clicking ok it should bring you to the page shown in the below figure, try to click Test Connection and it should popup with a success message. Click on Finish.



19. Return to the JPA page after having set up the connection and fill the page as shown in the picture below, then click Apply and Close.



20. Now everything should be alright if you have any error for missing libraries, download them and put them in the classpath of the project manually (we suggest to put it in the TomEE *lib* folder) and rebuild the project.

21. Finally load the Web project on the server and you should be redirected to the login page on the browser, we recommend not to use the Eclipse internal browser.

7. REFERENCES

- Git repository: https://github.com/duiliocirino/CirinoCocchia/
- https://www.eclipse.org/
- http://tomee.apache.org/
- https://www.eclipse.org/eclipselink/
- https://www.mysql.com/
- Mockito framework site

8. EFFORT SPENT

Duilio Cirino:

9-1	3	Customer UI creation
10-1	2	Manager/Employee UI creation
17-1	3	Customer Servlet Implementation
18-1	3	Manager servlet implementation
20-1	4	First revision of the UI/Servlet working
21-1	2	Rearranged servlets and pages division
28-1	4	Second revision of the UI/Servlet working
30-1	3	Realigned with the Business Logic tier modification and further corrections of errors and bugs in UI and servlets
31-1	2	Unit testing start
1-2	4	Further take on unit testing
2-2	4	Further take on unit testing
3-2	4	Unit testing completed
4-2	4	Fixing bugs
5-2	7	Fixing bugs
6-2	7	Fixing bugs
7-2	10	Finished

		documentation and fixed some project code

Lorenzo Cocchia:

9-1	1,30	Database creation
10-1	2	JavaBeans created
11-1	1	Introduction
17-1	2.30	Implementation of model services
22-1	2	Implementation of TimeEstimation Module
23-1	4	Finished to implement Reservation Management component
24-1	3	Implementation of Account and Grocery Management components
25-1	3	Implementation of Search component and revision of some entity
27-1	7	Tests on model, Account Management and Grocery Handler Management
28-1	4	Reservation handler unit tests
29-1	4	Reservation handler unit tests finished
30-1	4	Integration testing

31-1	4	Integration testing
1-2	4	Integration testing
3-2	4	Integration testing finished
4-2	7	Documentation
5-2	4	Fixing bugs
6-2	4	Fixing bugs and documentation
7-2	10	Last fixes on the project and documentation uploaded