Java EE 6 Hands-on Lab
Servlets 3.0 and
Java Persistence API 2.0 (JPA2)

Arun Gupta
blogs.sun.com/arungupta, @arungupta
Oracle Corporation

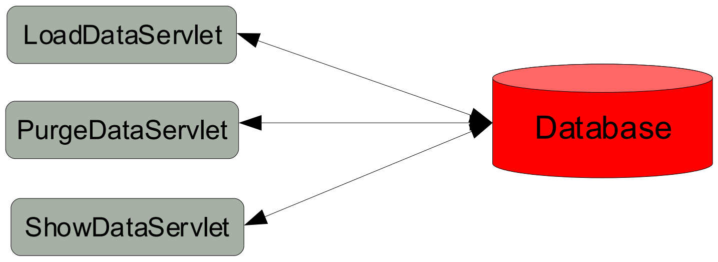
Table of Contents

1.0 Introduction

This hands-on lab provide detailed instructions on how to use some of the Servlets 3.0 and Java Persistence API 2.0 features with NetBeans 6.9.1 and GlassFish Open Source Edition 3.0.1.

2.0 Problem Statement

This hands-on lab builds a Web application that displays frequent flyer miles of different passengers. The application consists of three Servlets as shown below.



LoadDataServlet loads the initial data in the database, PurgeDataServlet deletes all data in the database, and ShowDataServlet shows the existing data from the database based upon user input. JPA entities for the database model are generated using NetBeans wizards and are used within these servlets.

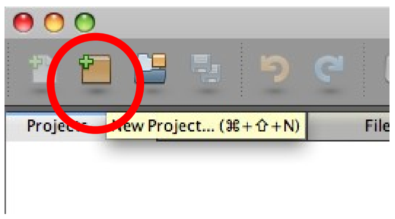
LoadData and PurgeData servlets are using Enterprise Java Beans for the transactional nature of the business logic. ShowData servlet use typesafe Criteria API to query the database and display all the users and their frequent flyer miles logged on each airline.

Even though Servlets are used for displaying the status messages but a more appropriate technology like Java Server Faces 2 should be used in a real-life application. Please consider taking Java Server Faces 2 and Context & Dependency Injection hands-on lab.

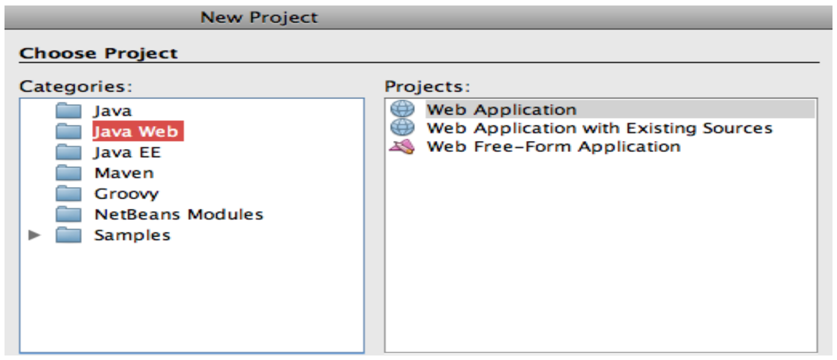
3.0 Build the template Web application

This step will build the template Web application.

3.1 In NetBeans IDE, create a new Web application by selecting the “New Project” icon as shown below:



3.2 Choose “Java Web” as category and “Web Applications” as Projects as shown below:



New Project

Choose Project

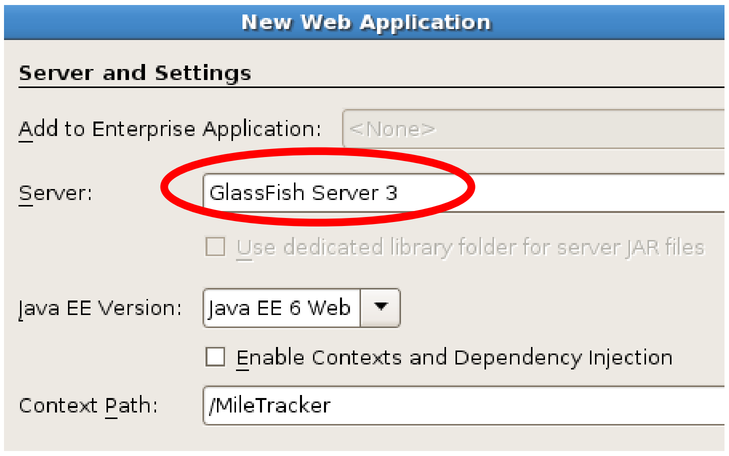
Categories:
- Java
- Java Web
- Java EE
- Maven
- Groovy
- NetBeans Modules
- Samples

Projects:
- Web Application
- Web Application with Existing Sources
- Web Free-Form Application

and click on “Next>”.

3.3 Specify the project name as “MileTracker” and click on “Next>”.

3.4 Choose the pre-configured “GlassFish Server 3” as the Server as shown and click on “Finish”.



New Web Application

Server and Settings

Add to Enterprise Application: <None>

Server: GlassFish Server 3

☐ Use dedicated library folder for server JAR files

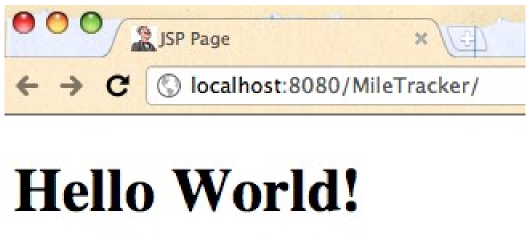
Java EE Version: Java EE 6 Web

☐ Enable Contexts and Dependency Injection

Context Path: /MileTracker

This generates a template Web project. Notice that there is no “web.xml” as it is optional in most of the common cases.

3.6 Right-click on the project and select “Run”. This will start the chosen GlassFish server, deploy the Web application on the server, opens the default web browser, and displays “http://localhost:8080/MileTracker/index.jsp”. The default page looks like as shown here.



JSP Page

localhost:8080/MileTracker/

Hello World!

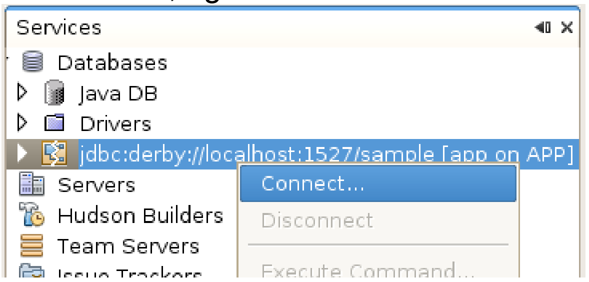
Note, even though “index.jsp” is not displayed in the URL window, this file is displayed as default.

A display of this page ensures that the project is successfully created.

4.0 Create Database Tables

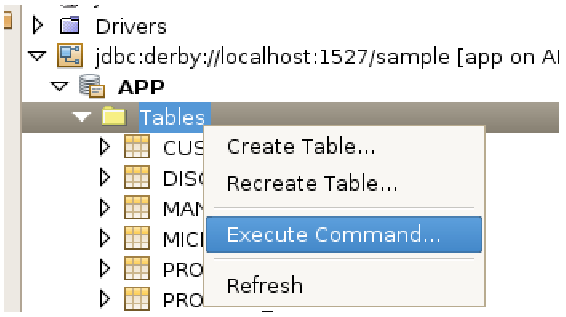
This section will create the database tables used to capture the data for our application, create servlets and backing EJBs to populate the database. It will use a pre-configured JDBC resource with the name “jdbc/sample” for creating the tables.

4.1 In NetBeans IDE “Services” tab, expand “Databases” node, right-click on “jdbc:derby://localhost:1527/sample” database connection, and select “Connect...” as shown.



This will start the database, if not already started, and connect to this pre-configured JDBC resource.

4.2 Expand the connected node, expand “APP”, expand “Tables” to see the list of default tables already existing in this schema. Right-click on “Tables” and select “Execute Command...”.

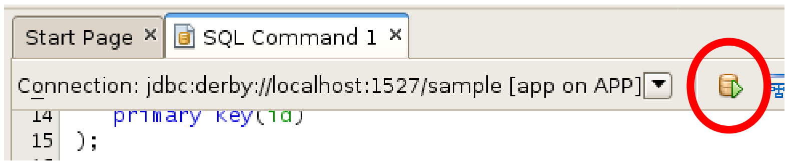


Enter the following DDL to generate the required tables:

```
create table passenger (
    id integer not null generated always as identity,
    name varchar(20),
    age integer,
    street varchar(30),
    city varchar(20),
    country varchar(20),
    primary key(id)
);

create table airline (
    id integer not null generated always as identity,
    name varchar(20),
    primary key(id)
);

create table logged_miles (
    passenger_id integer,
    airline_id integer,
    miles integer,
    primary key(passenger_id, airline_id)
);

alter table logged_miles
    add constraint pid_fk
    foreign key (passenger_id) references passenger(id);
```

```
alter table logged_miles
    add constraint aid_fk
    foreign key (airline_id) references airline(id);
```

And hit the green arrow button to execute this SQL as shown.

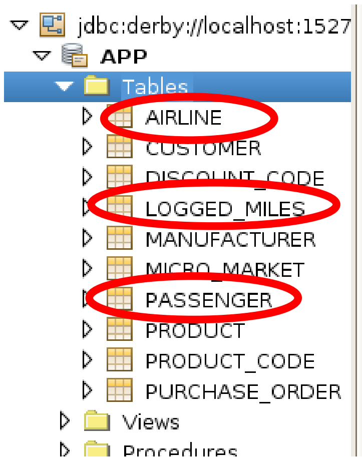


Connection: jdbc:derby://localhost:1527/sample [app on APP]

```
    primary key(id)
);
```

This DDL generates three sample tables “passenger”, “airline”, “logged_miles”. The “passenger” table stores the basic details about passenger, “airline” table stores basic details about different airlines, and “logged_miles” is a join table that stores how many miles each passenger has logged on different airlines. “logged_miles” has foreign keys referencing to the primary keys of “passenger” and “airline” tables.

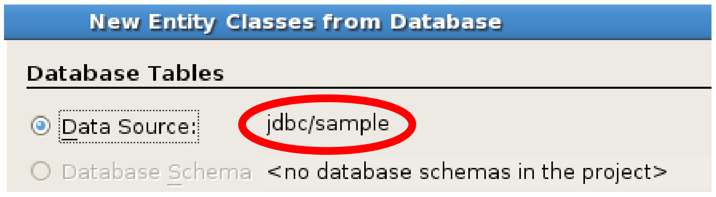
In the “Services” tab, right-click on “Tables” and click on “Refresh” to see the updated list of tables as shown.



5.0 Create JPA Entities

This section will use NetBeans wizards to generate the JPA entities and customize the entities to make them more intuitive for Java developers.

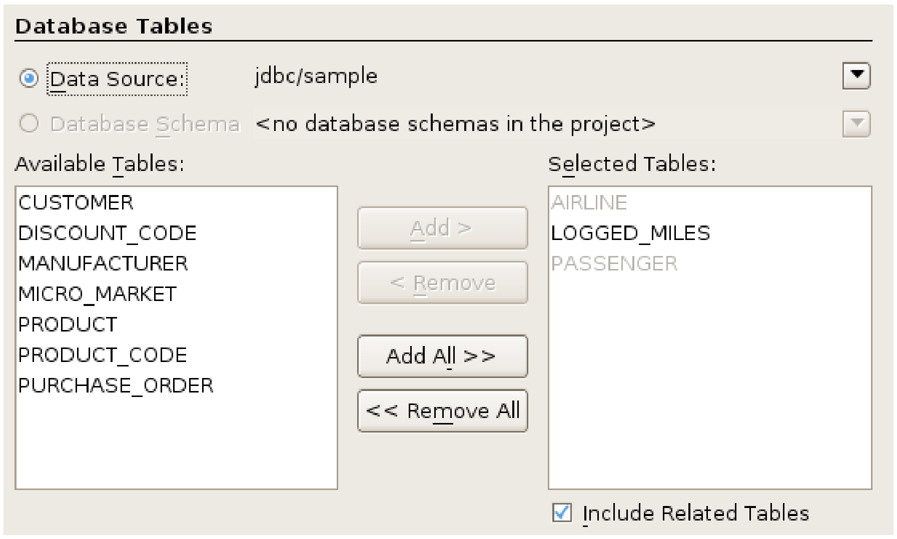
5.1 In NetBeans IDE, right-click on “MileTracker” project and select “New”, “Entity Classes from Database...”. Choose “jdbc/sample” as the Data Source from the drop down list box as shown. This will show all the tables from this data source.



Data Source: jdbc/sample

Database Schema <no database schemas in the project>

5.2 Select “LOGGED_MILES” table from the “Available Tables” and click “Add>”. Notice, “PASSENGER” and “AIRLINE” tables are automatically selected because of foreign key references and “Include Related Tables” checkbox as shown:



Database Tables

● Data Source: jdbc/sample

○ Database Schema <no database schemas in the project>

Available Tables:	Selected Tables:
CUSTOMER	AIRLINE
DISCOUNT_CODE	LOGGED_MILES
MANUFACTURER	PASSENGER
MICRO_MARKET	
PRODUCT	
PRODUCT_CODE	
PURCHASE_ORDER	

☑ Include Related Tables

and click on “Next>”.

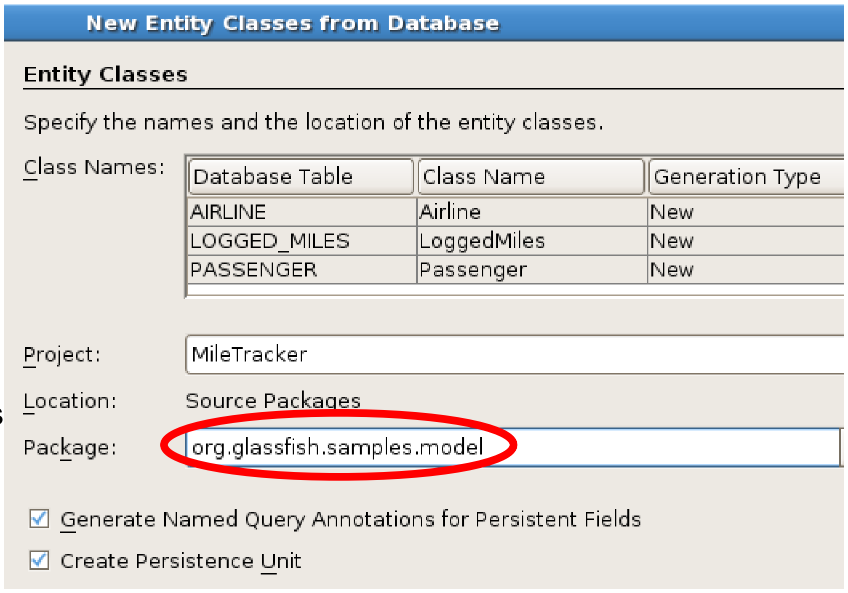
5.3 Enter the package name “org.glassfish.samples.model” as shown below:

The mapped class names are shown in “Class Name” column and can be changed here, if needed.

The first check box allows NetBeans IDE to generate multiple @NamedQuery for convenience.

The second check box generates the “persistence.xml” required by JPA.

Click on “Finish” to complete the entity generation.



New Entity Classes from Database

Entity Classes

Specify the names and the location of the entity classes.

Class Names:

Database Table	Class Name	Generation Type
AIRLINE	Airline	New
LOGGED_MILES	LoggedMiles	New
PASSENGER	Passenger	New

Project: MileTracker

Location: Source Packages

Package: org.glassfish.samples.model

☑ Generate Named Query Annotations for Persistent Fields

☑ Create Persistence Unit

Make sure to explicitly deploy the project again by right-clicking on project and selecting “Deploy” otherwise you'll encounter the NetBeans issue “https://netbeans.org/bugzilla/show_bug.cgi?id=190987”.

5.4 Understand and Customize JPA entities

This section will customize the generated JPA entities for a more usable Java class.

5.4.1 In NetBeans IDE, expand “Source Packages”, and double-click “Airline.java”, change the “toString()” method to the one shown below:

```
@Override
public String toString() {
    return name;
}
```

Also look at the generated class-level @NamedQuery annotations.

Add the following convenience constructor to this class:

```
public Airline(String name) {
    this.name = name;
}
```

The Collection<LoggedMiles> field is marked with @OneToMany annotation indicating that one Airline can have many LoggedMiles entities.

5.4.2 Edit “LoggedMiles.java” and add the following convenience constructor:

```
public LoggedMiles(int passengerId, int airlineId, Integer miles) {
    this(passengerId, airlineId);
    this.miles = miles;
}
```

Notice the following points:

- @EmbeddedId annotation indicates that the primary key is an embeddable class.
- “passenger” and “airline” fields are marked with @JoinColumn creating a join with the appropriate tables.

5.4.3 Edit “Passenger.java” and change the class structure to introduce an Embeddable class for street, city, and country fields as these are logically related entities.

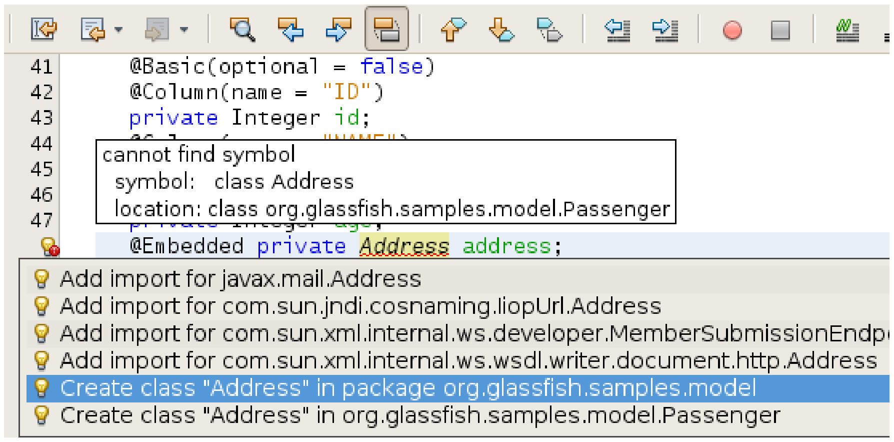
5.4.3.1 Replace the following code:

```
@Column(name = “STREET”)
private String street;
@Column(name = “CITY”)
private String city;
@Column(name = “COUNTRY”)
private String country;
```

with

```
@javax.persistence.Embedded private Address address;
```

Use the yellow bulb in the left bar to create a new class in the current package as shown below:



```
@Basic(optional = false)
@Column(name = "ID")
private Integer id;
@Embedded private Address address;
```

cannot find symbol
symbol: class Address
location: class org.glassfish.samples.model.Passenger

Add import for javax.mail.Address
Add import for com.sun.jndi.cosnaming.IiopUrl.Address
Add import for com.sun.xml.internal.ws.developer.MemberSubmissionEndp
Add import for com.sun.xml.internal.ws.wsdl.writer.document.http.Address
Create class "Address" in package org.glassfish.samples.model
Create class "Address" in org.glassfish.samples.model.Passenger

@Embedded annotation says that this field's value is an instance of an embeddable class.

5.4.3.2 Change “Address” class so that it is a public class, annotated with @Embeddable such that it can be used as embeddable class later, and implements Serializable interface by making the change as shown below:

```
@javax.persistence.Embeddable
public class Address implements java.io.Serializable
```

5.4.3.3 In “Address” class, add three fields as:

```
private String street;
private String city;
private String country;
```

and add getter/setters for each field. Also add the default and a convenience constructor as shown:

```
public Address() { }

public Address(String street, String city, String country) {
    this.street = street;
    this.city = city;
    this.country = country;
}
```

5.4.3.4 In “Passenger.java”, remove the getter/setter for previously removed fields “street”, “city”, and “country” and a new getter/setter for “address” field as:

```
public Address getAddress() { return address; }
public void setAddress(Address address) { this.address = address; }
```

5.4.3.5 In “Passenger.java”, change the @NamedQuery to reflect the nested structure for Address by editing “Passenger.findByStreet”, “Passenger.findByCity”, “Passenger.findByCountry” such that “p.street”, “p.city”, and “p.country” is replaced with “p.address.street”, “p.address.city”, “p.address.country” respectively.

5.5 Edit “Passenger.java” and add the following convenience constructor:

```
public Passenger(String name, Integer age, Address address) {
    this.name = name;
    this.age = age;
    this.address = address;
}
```

The Collection<LoggedMiles> field is marked with @OneToMany annotation indicating that one Passenger can have many LoggedMiles entities.

6.0 Loading data into the Database using LoadDataServlet

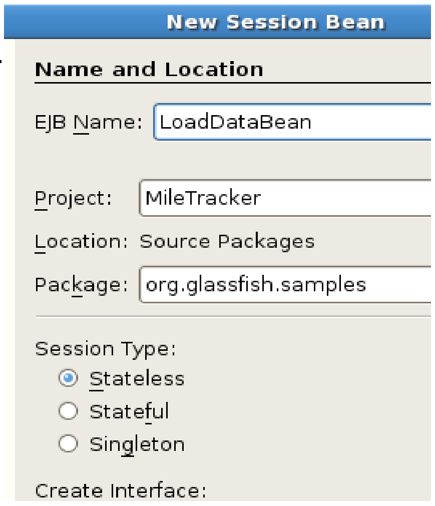
This section will create a Servlet that will load the initial set of data into the tables. The Servlet will use an Enterprise JavaBean to populate the database and thus use Container Managed Transactions to show the simplicity of Java EE 6 platform.

6.1 Right-click on the project, select “New”, “Servlet...”. Enter the class name as “LoadDataServlet”, package as “org.glassfish.samples”, and click on “Finish”.

Expand “Web Pages”, “WEB-INF” and notice there is no “web.xml”. Instead all the required information is generated as annotations in the source file using @WebServlet as shown:

```
@WebServlet(name="LoadData", urlPatterns={"/LoadDataServlet"})
public class LoadDataServlet extends HttpServlet {
```

6.2 Create a new EJB to store all the data in the tables. Right-click on “org.glassfish.samples” package, select “New”, “Session Bean...”, specify the EJB Name as “LoadDataBean”, take all defaults, and click on “Finish” as shown. This will create a stateless EJB.



EJB Name: LoadDataBean
Project: MileTracker
Location: Source Packages
Package: org.glassfish.samples
Session Type: [✓ Stateless] Stateful Singleton
Create Interface:

6.3 EJBs are not re-entrant and so we can inject an instance of EntityManager in this bean, instead of EntityManagerFactory, as shown:

```
@PersistenceContext
EntityManager em;
```

6.4 In this bean, create a new business method, create some entities and persist them to load data in the tables as shown below:

```
public void loadData() {
    Airline united = new Airline("United");
    Airline continental = new Airline("Continental");
    Airline lufthansa = new Airline("Lufthansa");

    em.persist(united);
    em.persist(continental);
    em.persist(lufthansa);

    Passenger john = new Passenger("John", 38, new Address("123, Second Street",
"CA", "USA"));
    Passenger bill = new Passenger("Bill", 34, new Address("456, Orchard Road",
"London", "UK"));
    Passenger nichole = new Passenger("Nichole", 32, new Address("789, Main St",
"Sydney", "Australia"));
    em.persist(john);
    em.persist(bill);
    em.persist(nichole);

    em.flush();

    LoggedMiles lm1 = new LoggedMiles(john.getId(), united.getId(), 2000);
    LoggedMiles lm2 = new LoggedMiles(john.getId(), continental.getId(), 5000);

    LoggedMiles lm3 = new LoggedMiles(bill.getId(), continental.getId(), 6000);
    LoggedMiles lm4 = new LoggedMiles(bill.getId(), lufthansa.getId(), 6000);

    LoggedMiles lm5 = new LoggedMiles(nichole.getId(), united.getId(), 2000);
    LoggedMiles lm6 = new LoggedMiles(nichole.getId(), continental.getId(), 4000);
    LoggedMiles lm7 = new LoggedMiles(nichole.getId(), lufthansa.getId(), 3000);

    em.persist(lm1);
    em.persist(lm2);
    em.persist(lm3);
    em.persist(lm4);
    em.persist(lm5);
    em.persist(lm6);
    em.persist(lm7);
}
```

Fix the imports. On certain machines, the copy/paste creates redundant curly braces, {}, at the end of the first Passenger instantiation. Please verify to make sure that is not the case in your environment.

The code instantiates several JPA entities with data and calls “em.persist” to store these entities in the database. The method “loadData” is a public method in EJB and is executed in a transactional context. This ensures that the transaction is committed at the end of method execution.

Notice, “em.flush()” is required after “Airline” and “Passenger” entities are created to ensure their “getId” method will work as expected. This method shows one simple way to load data in the database. Typically input is received from a front-end using a form and then loaded in the database.

6.5 Invoke EJB business method from Servlet

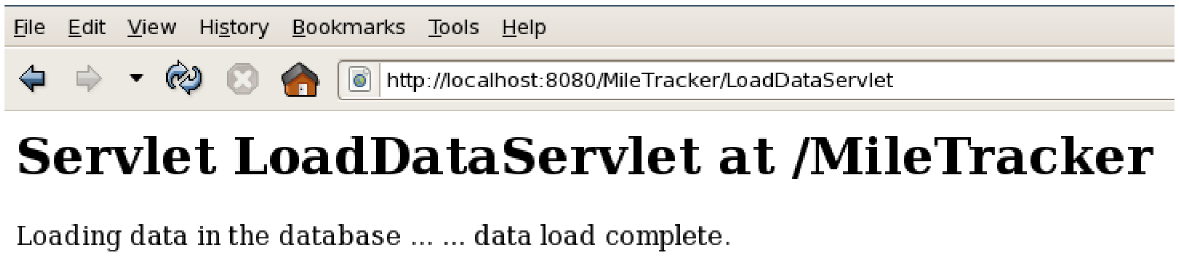
Inject this bean into Servlet using the following code:

```
@EJB LoadDataBean ejb;
```

Fix the imports. Uncomment code in the “try” block of “processRequest” method of “LoadDataServlet” and add code to invoke the business method. The updated “try” block looks like:

```
try {
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet LoadDataServlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Servlet LoadDataServlet at " + request.getContextPath () +
"</h1>");
    out.println("Loading data in the database ...");
    ejb.loadData();
    out.println("... data load complete.");
    out.println("</body>");
    out.println("</html>");
} finally {
```

6.6 Load “http://localhost:8080/MileTracker/LoadDataServlet” in a browser window to see the output as:

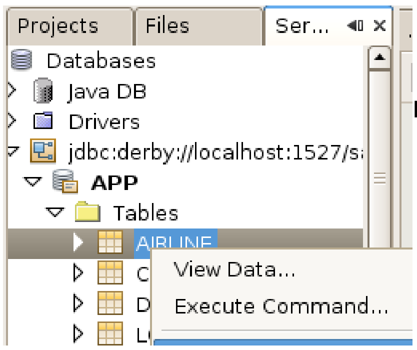


Servlet LoadDataServlet at /MileTracker

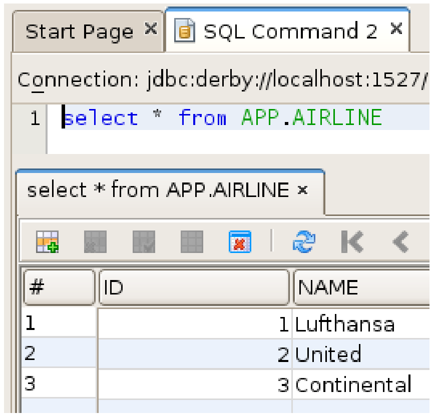
Loading data in the database ... ... data load complete.

This ensures that the entities created in the EJB are stored in the underlying tables.

6.7 In order to verify the data, go to “Services” tab in NetBeans IDE, expand “Databases”, “jdbc:derby://localhost:1527/sample”, “APP”, “Tables”, right-click on “AIRLINE”, and select “View Data...” as shown.



The output is as shown.



Connection: jdbc:derby://localhost:1527/

```
select * from APP.AIRLINE
```

select * from APP.AIRLINE

#	ID	NAME
1	1	Lufthansa
2	2	United
3	3	Continental

The data in other tables can be verified similarly as well.

7.0 Deleting data from the Database using PurgeDataServlet (OPTIONAL)

This section will explain how to delete data from the database in a Servlet. The Servlet will use an Enterprise JavaBean to delete data from database and thus use Container Managed Transactions.

7.1 Create a new servlet “PurgeDataServlet” in the package “org.glassfish.samples” following the steps described in section 6.1.

7.2 Create a new Stateless EJB as explained in section 6.2 and name it as “PurgeDataBean”. Inject an EntityManager as explained in section 6.3.

7.3 In this bean, create a new business method to delete all the data as shown below:

```
public void purgeData() {
    for (Object o : em.createNamedQuery("LoggedMiles.findAll").getResultList()) {
        em.remove(o);
    }
    for (Object o : em.createNamedQuery("Airline.findAll").getResultList()) {
        em.remove(o);
    }
    for (Object o : em.createNamedQuery("Passenger.findAll").getResultList()) {
        em.remove(o);
    }
}
```

This method uses pre-defined @NamedQuery to fetch all the rows and then delete them. The method is executed in a new transactional context.

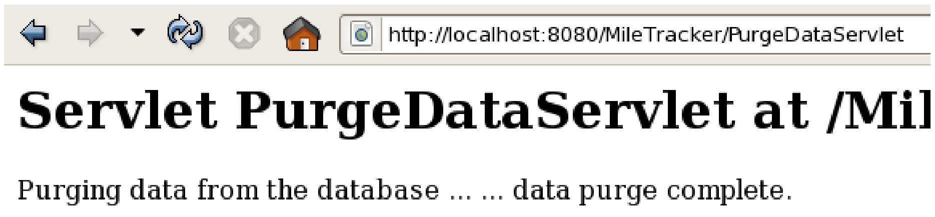
7.4 Inject this bean in “PurgeDataServlet” as

```
@EJB PurgeDataBean ejb;
```

and invoke it in “processRequest” method as shown below:

```
try {
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet PurgeDataServlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Servlet PurgeDataServlet at " + request.getContextPath () +
"</h1>");
    out.println("Purging data from the database ...");
    ejb.purgeData();
    out.println("... data purge complete.");
    out.println("</body>");
    out.println("</html>");
} finally {
```

7.5 Load “http://localhost:8080/MileTracker/PurgeDataServlet” in a browser window to see the



```
http://localhost:8080/MileTracker/PurgeDataServlet
```

Servlet PurgeDataServlet at /Mil

Purging data from the database ... ... data purge complete.

output as:

This ensures that all rows from all the tables for this application are deleted.

8.0 Displaying data from the database using ShowDataServlet

This section will explain how to read data from the database table in a Servlet using JPA2 type-safe Criteria API.

8.1 In “Web Pages”, edit “index.jsp” and add the following fragment

```
<form action="ShowDataServlet" method="post">
    Name: <input type="text" name="name">
    <button type="submit">Show Miles</button>
</form>
```

This HTML form will take user's input for a name and post the input to “ShowDataServlet”.

8.2 Create a new servlet “ShowDataServlet” following the steps described in section 6.1. Make sure to inject EntityManagerFactory.

8.3 Edit “processRequest” method such that the “try” block looks like:

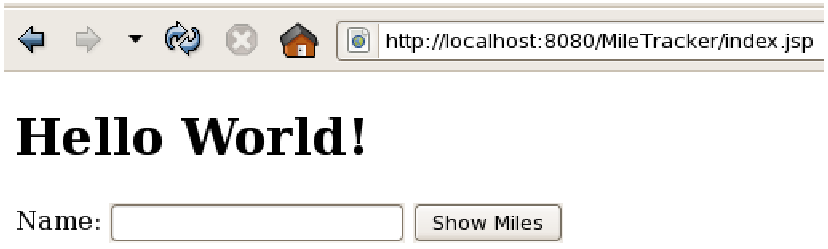
```
try {
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet ShowDataServlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Servlet ShowDataServlet at " + request.getContextPath () +
"</h1>");
    String name = request.getParameter("name");
    if (name != null) {
        EntityManager em = emf.createEntityManager();
        CriteriaBuilder builder = em.getCriteriaBuilder();
        CriteriaQuery<Passenger> criteriaQuery =
builder.createQuery(Passenger.class);

        // FROM clause
        Root<Passenger> root = criteriaQuery.from(Passenger.class);

        // WHERE clause
        Predicate condition = builder.equal(root.get("name"), name);
        criteriaQuery.where(condition);

        // FIRE query
        TypedQuery<Passenger> query = em.createQuery(criteriaQuery);

        // PRINT result
        List<Passenger> passengers = query.getResultList();
        for (Passenger p : passengers) {
            out.println("<b>" + p.getName() + "</b><br/>");
            for (LoggedMiles miles : p.getLoggedMilesCollection()) {
                out.println(miles.getAirline() + ": " + miles.getMiles() +
"<br/>");
            }
            out.println("<br/>");
        }
    }
    out.println("</body>");
    out.println("</html>");
} finally {
```

The code fragment has the boilerplate code generated by NetBeans. It uses JPA2 type-safe Criteria API, within the “if” block, to query the database. Notice that we are using String-based navigation here in the WHERE clause and so loose the complete type-safety. In real-life it is expected to generate Metamodel classes using either an Annotation Processor or IDEs and use them for complete type-safety.

8.4 Inject an EntityManagerFactory in the Servlet as:

```
@PersistenceUnit EntityManagerFactory emf;
```

Fix the imports.
8.4 Load "http://localhost:8080/MileTracker/index.jsp" in a browser window and now it looks like as shown. The only valid names are "John", "Bill", and "Nichole".

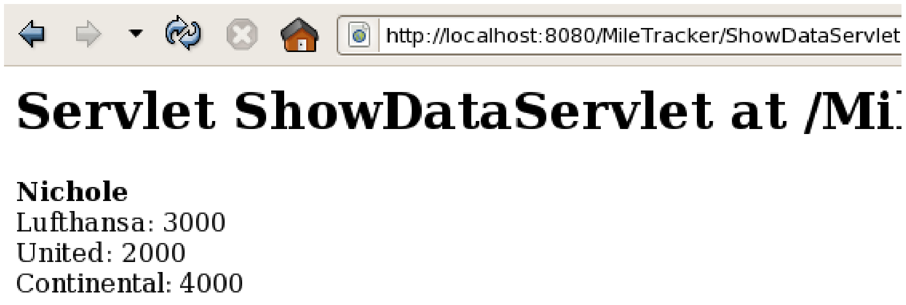


http://localhost:8080/MileTracker/index.jsp

Hello World!

Name: [ ] Show Miles

Enter "Nichole" in the text box and click on "Show Miles" to see the output as:



http://localhost:8080/MileTracker/ShowDataServlet

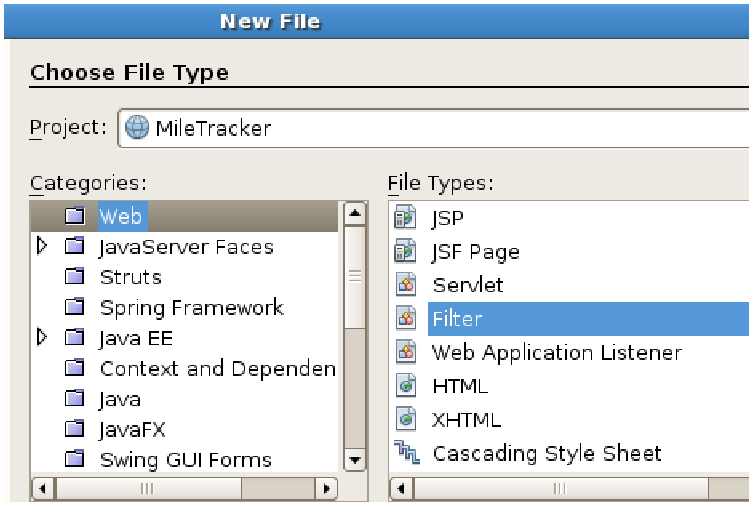
Servlet ShowDataServlet at /Mi

Nichole
Lufthansa: 3000
United: 2000
Continental: 4000

9.0 Logging using Servlet Filters (OPTIONAL)

This section will add a ServletFilter to address the cross-cutting concern of logging for one particular servlet.

9.1 Right-click on "org.glassfish.samples" package, select "New", "Other...", select "Web", and "Filter" as shown, and click on "Next>".



New File

Choose File Type

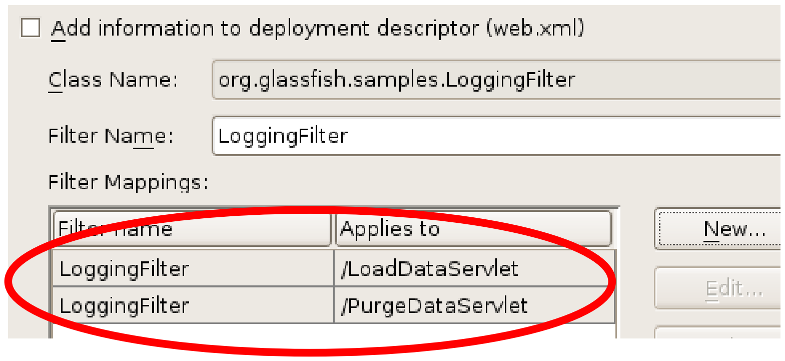
Project: MileTracker

Categories: Web, JavaServer Faces, Struts, Spring Framework, Java EE, Context and Dependen, Java, JavaFX, Swing GUI Forms

File Types: JSP, JSF Page, Servlet, Filter, Web Application Listener, HTML, XHTML, Cascading Style Sheet

9.2 Specify the class name as "LoggingFilter", take everything else as default and click on "Next>".

9.3 Click on "New..." and add the URL as "/PurgeDataServlet". Add "/LoadDataServlet" following the same steps and then click on "OK". The completed screen looks like as shown below:


☐ Add information to deployment descriptor (web.xml)

Class Name: org.glassfish.samples.LoggingFilter

Filter Name: LoggingFilter

Filter Mappings:

Filter name	Applies to
LoggingFilter	/LoadDataServlet
LoggingFilter	/PurgeDataServlet

Note, this filter will be applied only to "/PurgeDataServlet" and "/LoadDataServlet" and none of the information about ServletFilter will be added to the deployment descriptor.

Click on "Finish".

The generated class definition looks like:

```
@WebFilter(filterName="SecurityFilter", urlPatterns={"/PurgeDataServlet",
"/LoadDataServlet"})
public class LoggingFilter implements Filter {
```

9.4 In "doBeforeProcessing", uncomment the block comment and fix the import for "Enumeration". Edit the first line of this method so that it looks like:

```
if (debug) log(((HttpServletRequest)request).getRequestURL() + " -
SecurityFilter:DoBeforeProcessing");
```

This will print a debug statement with the complete request URL and then log the request parameters before the servlet is invoked.

9.5 In "doAfterProcessing", uncomment the first block comment. Edit the first line of this method so that it looks like:

```
if (debug) log(((HttpServletRequest)request).getRequestURL() + " -
SecurityFilter:DoAfterProcessing");
```

This will print a debug statement with the complete request URL and log the request attributes after the request has been processed.

9.6 Open "http://localhost:8080/MileTracker/PurgeDataServlet" in a browser window to see the output shown in section 7.5. In addition, the GlassFish log in NetBeans IDE shows the following entries:

```
INFO: PWC1412: WebModule[/MileTracker]
ServletContext.log():SecurityFilter:doFilter()
INFO: PWC1412: WebModule[/MileTracker]
ServletContext.log():http://localhost:8080/MileTracker/PurgeDataServlet -
SecurityFilter:DoBeforeProcessing
INFO: PWC1412: WebModule[/MileTracker]
ServletContext.log():http://localhost:8080/MileTracker/PurgeDataServlet -
SecurityFilter:DoAfterProcessing
```

The specific log statements are highlighted in bold. Similar log statements are also shown when "http://localhost:8080/MileTracker/LoadDataServlet" is viewed in browser.

These log statements are not displayed when the "Show Miles" button is clicked on "index.jsp" which in turn invokes ShowDataServlet.

10.0 Conclusion

This hands-on lab created a simple application demonstrating key features of Servlets 3.0 and Java Persistence API 2.0, namely:

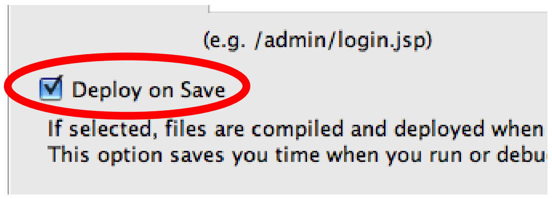
- Annotation-based and Deployment Descriptor-free Servlets and Servlet Filter
- Injecting EJB in a Servlet
- Creating JPA entities using NetBeans wizards
- JPA NamedQuery with JPQL to access database tables
- JPA Type-safe Criteria API to access database tables
- Reading, Storing, and Deleting Entities from Servlets and EJBs

Hopefully this has raised your interest enough in trying out Java EE 6 applications using GlassFish and NetBeans.

Send us feedback at users@glassfish.dev.java.net.

11.0 Troubleshooting

11.1 The project is getting deployed to GlassFish every time a file is saved. How can I disable/enable that feature ?


(e.g. /admin/login.jsp)
[✓] Deploy on Save
If selected, files are compiled and deployed when
This option saves you time when you run or debu

This feature can be enabled/disable per project basis from the Properties window. Right-click on the project, select "Properties", choose "Run" categories and select/unselect the checkbox "Deploy on Save" to enable/disable this feature.

11.2 How can I show the SQL queries issued to the database ?

In NetBeans IDE, expand “Configuration Files”, edit “persistence.xml” and replace:

```
<properties/>
```

with

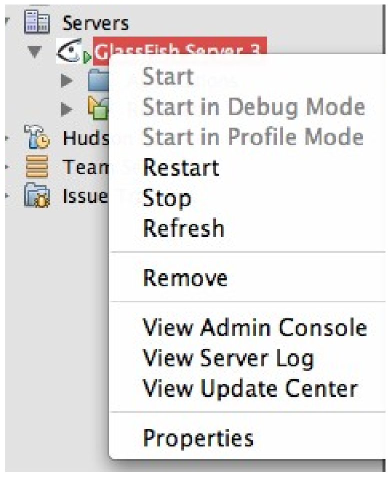
```
<properties>
    <property name="eclipselink.logging.level" value="FINE" />
</properties>
```

Then ShowDataServlet, as explained in section 8.0, shows the following output in GlassFish logs within NetBeans:

```
FINE: SELECT ID, NAME, AGE, STREET, CITY, COUNTRY FROM PASSENGER WHERE (NAME = ?)
        bind => [Nichole]
FINE: SELECT MILES, PASSENGER_ID, AIRLINE_ID FROM LOGGED_MILES WHERE (PASSENGER_ID
= ?)
        bind => [43]
FINE: SELECT ID, NAME FROM AIRLINE WHERE (ID = ?)
        bind => [43]
FINE: SELECT ID, NAME FROM AIRLINE WHERE (ID = ?)
        bind => [44]
FINE: SELECT ID, NAME FROM AIRLINE WHERE (ID = ?)
        bind => [45]
```

11.3 How can I start/stop/restart GlassFish from within the IDE ?

In the “Services” tab, right-click on “GlassFish Server 3”, and all the commands to start, stop, and restart are available from the pop-up menu. The server log can be viewed by clicking on “View Server Log” and web-based administration console can be seen by clicking on “View Admin Console”.



12.0 Possible Improvements

Do you feel motivated and would like to try some enhancements to the application ? Here is one:

12.1 Use ServletFilter for HTTP Basic Authentication.

12.2 Generate Metamodel for JPA entities and write completely type-safe queries using Criteria API.

12.3 Edit “index.jsp” to accept new Passenger information and add to the database.

13.0 Completed Solutions

The completed solution is available as a NetBeans project in the home directory of the VirtualBox instance by the directory name “MileTracker”. Open the project in NetBeans, browse through the source code, and enjoy!

The completed solution can also be downloaded from http://blogs.oracle.com/arungupta/resource/MileTracker.zip.