Parallel and Distributed Software Systems : Java Enterprise Applications and Web Services

November, 11th 2014
pds@intec.ugent.be

Objectives of this tutorial:

- Explain how to develop, deploy and run Java Enterprise applications using the Netbeans IDE.
- Give an overview of example Java EE applications with enterprise beans (entities, session beans)
- Introduce RESTful Web Services
- Introduce JSON

Before you start:

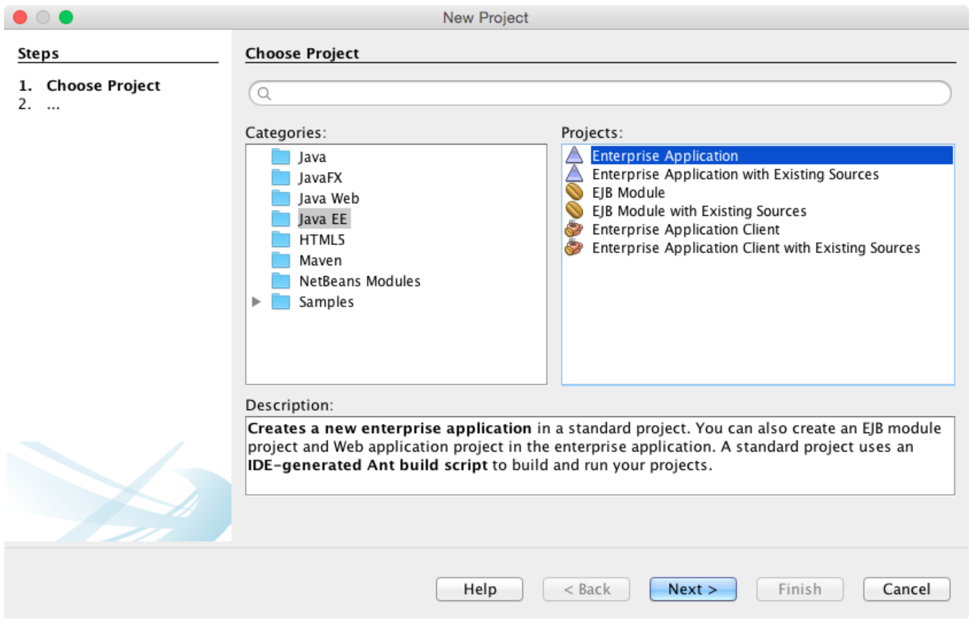
- The software used in this lab session is Netbeans IDE 7.4 (`https://netbeans.org/downloads/7.4/index.html`). Netbeans 7.4 is used in this tutorial and lab session, as the most recent version of Netbeans contains bugs when new Java databases are created.
- More information on Java EE 7 and GlassFish security is available on Minerva.

1 Java EE (Enterprise Edition) 7 Platform and Netbeans IDE 7.4

1.1 Creating a Java EE application

To create a Java EE IDE project you should do the following:

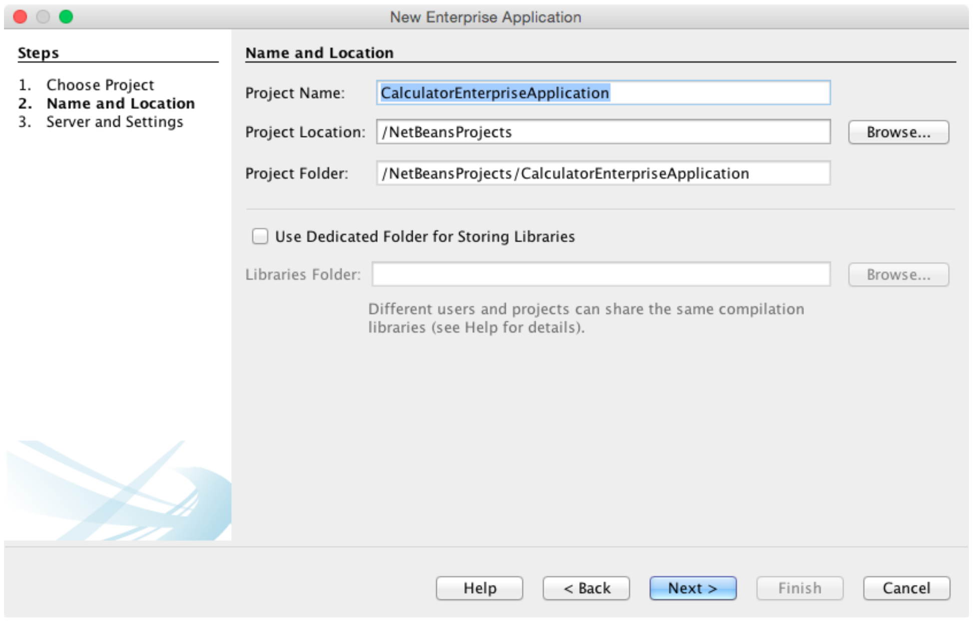
1. Start the NetBeans IDE.
2. In the IDE, choose File > New Project (or use the shortcut Ctrl-Shift-N).
3. In the New Project wizard, expand the Java EE category and select Enterprise Application as shown in the figure below. Then click Next.



Description:
Creates a new enterprise application in a standard project. You can also create an EJB module project and Web application project in the enterprise application. A standard project uses an **IDE-generated Ant build script** to build and run your projects.

Figure 1: Creating a new Java EE project

4. In the Name and Location page of the wizard, do the following (as shown in the figure below). In the Project Name field, type `CalculatorEnterpriseApplication`.



Project Name: CalculatorEnterpriseApplication
Project Location: /NetBeansProjects
Project Folder: /NetBeansProjects/CalculatorEnterpriseApplication
☐ Use Dedicated Folder for Storing Libraries
Libraries Folder:
Different users and projects can share the same compilation libraries (see Help for details).

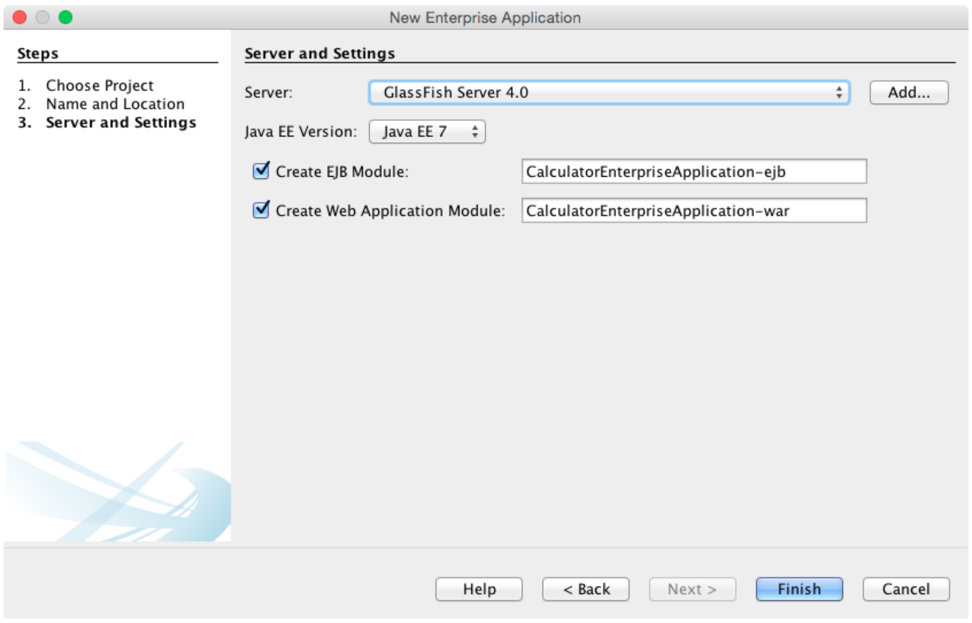
Figure 2: Creating a new Java EE project, step 2

5. Choose the Application Server and select the modules that should be created. The used Application Server is GlassFish Server 4.0. Select Java EE 7. You can select:

   (a) Create EJB Module, the creation of the Enterprise Beans module (EJB)

   (b) Create Web Application Module, the creation of a web container (war)

   By default, the EJB Module and Web Application module are selected.



Server: GlassFish Server 4.0

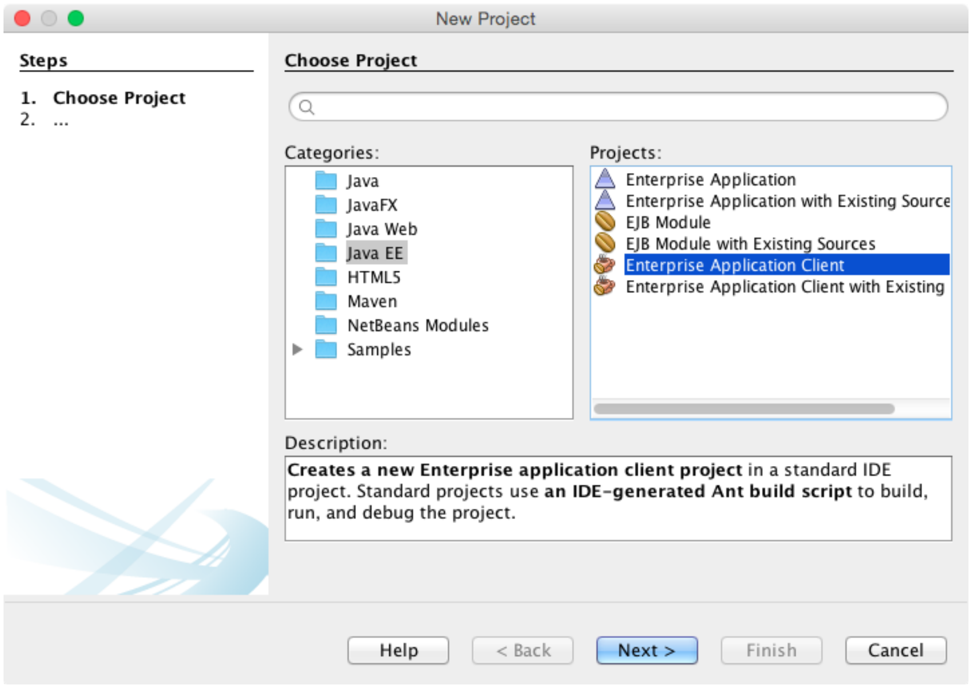
Java EE Version: Java EE 7

☑ Create EJB Module: CalculatorEnterpriseApplication-ejb

☑ Create Web Application Module: CalculatorEnterpriseApplication-war

Figure 3: Creating a new Java EE project, step 3

6. Click Finish.
7. Now, create an Application Client Module by creating a new Enterprise Application Client Project.

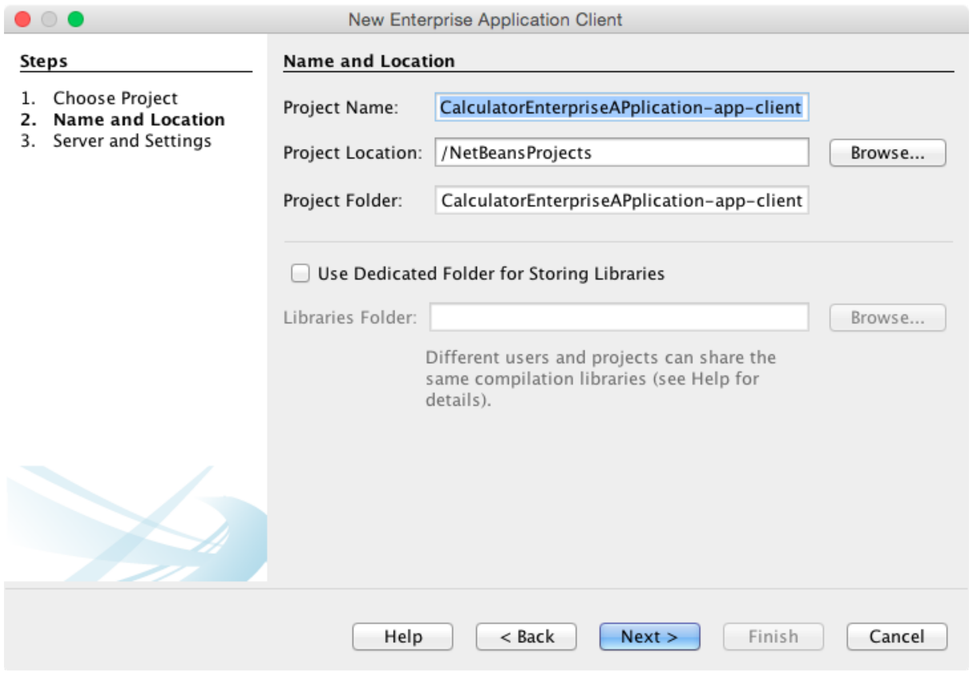


Description:

Creates a new Enterprise application client project in a standard IDE project. Standard projects use **an IDE-generated Ant build script** to build, run, and debug the project.

Figure 4: Creating a new Application Client project

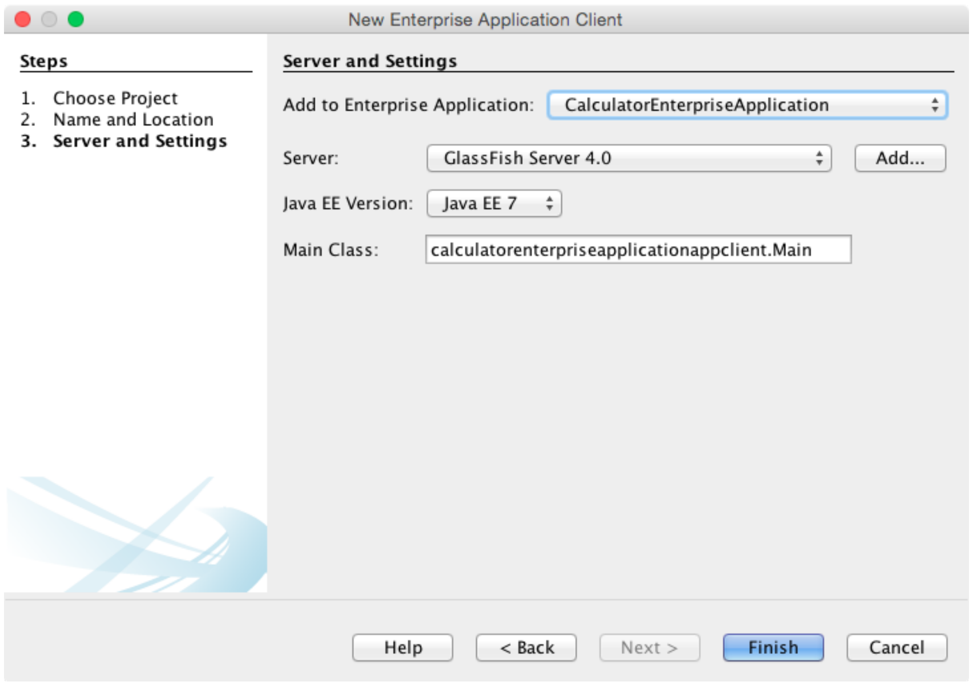
8. Call it `CalculatorEnterpriseApplication-app-client`.



Project Name: CalculatorEnterpriseAPplication-app-client
Project Location: /NetBeansProjects
Project Folder: CalculatorEnterpriseAPplication-app-client
Use Dedicated Folder for Storing Libraries
Libraries Folder:
Different users and projects can share the same compilation libraries (see Help for details).

Figure 5: Creating a new Application Client project, step 2

9. Now, add the Application Client to the Java EE project.



Add to Enterprise Application: CalculatorEnterpriseApplication
Server: GlassFish Server 4.0
Java EE Version: Java EE 7
Main Class: calculatorenterpriseapplicationappclient.Main

Figure 6: Creating a new Application Client project, step 3

At this moment an empty Java Enterprise application has been created. The modules are shown in the projects window of the Netbeans IDE (as shown in Figure 7). Now, the creation of the business logic (in the session beans) or persistence (in the entities) can be started.

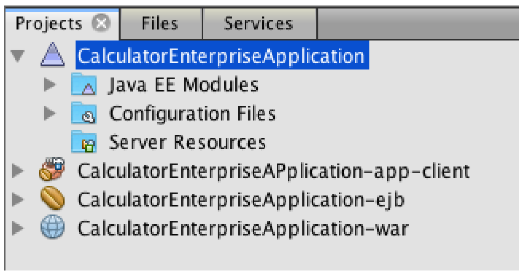

Projects
Files
Services
CalculatorEnterpriseApplication
Java EE Modules
Configuration Files
Server Resources
CalculatorEnterpriseAPplication-app-client
CalculatorEnterpriseApplication-ejb
CalculatorEnterpriseApplication-war

Figure 7: Overview of the created modules in the Netbeans IDE

1.2 The Application Server “GlassFish Server 4.0”

GlassFish Server 4.0 is an open source Application Server project, for building Java EE 7 applications. It is based on source code for Sun Java System Application Server PE9, donated by Sun Microsystems and TopLink persistence code, donated by Oracle (More info: `https://glassfish.dev.java.net`).
Databases and Servers can be found in the Services tab. Right-click GlassFish Server 4.0 and click Start. The Application Server will be started. This can take several minutes.

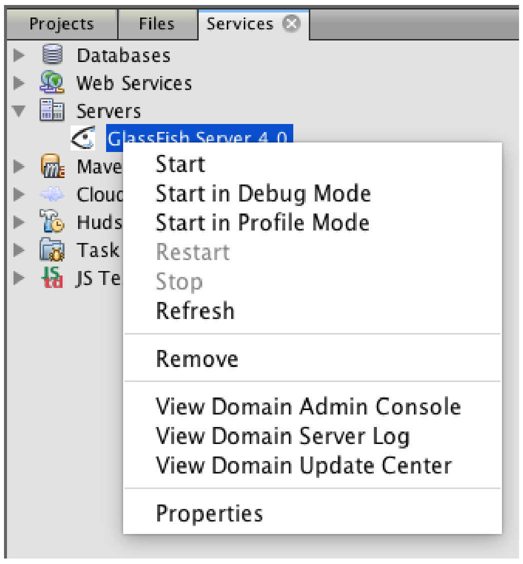
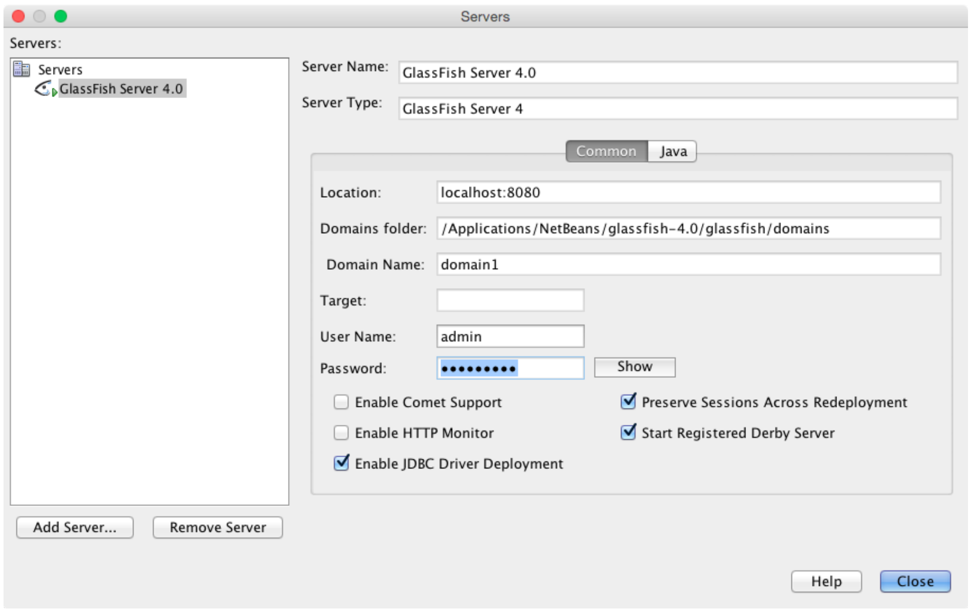


Figure 8: Services window in the Netbeans IDE

The configuration of the GlassFish Application Server is accessible through the services window. If you select ‘View Admin Console’ you get access to the detailed configuration (or browse to `http://localhost:4848/`. NetBeans 7.4 does not ask you to set a password for the GlassFish Application Server, you can find the password when you right-click on the Server and choose Properties. There, you can make your password visible, as shown in Figure 9. Do not change the password in the GUI as this will not reflect within the configuration files of the Server itself.



Server Name: GlassFish Server 4.0
Server Type: GlassFish Server 4
Location: localhost:8080
Domains folder: /Applications/NetBeans/glassfish-4.0/glassfish/domains
Domain Name: domain1
Target:
User Name: admin
Password: •••••••••

- [ ] Enable Comet Support
- [ ] Enable HTTP Monitor
- [x] Enable JDBC Driver Deployment
- [x] Preserve Sessions Across Redeployment
- [x] Start Registered Derby Server

Figure 9: Location of the GlassFish password

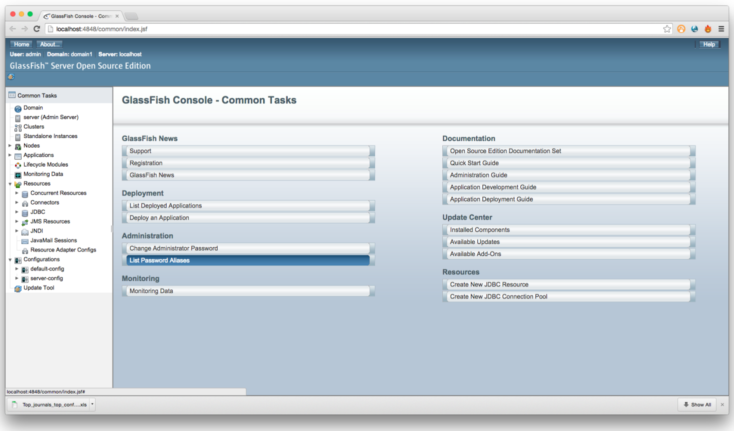
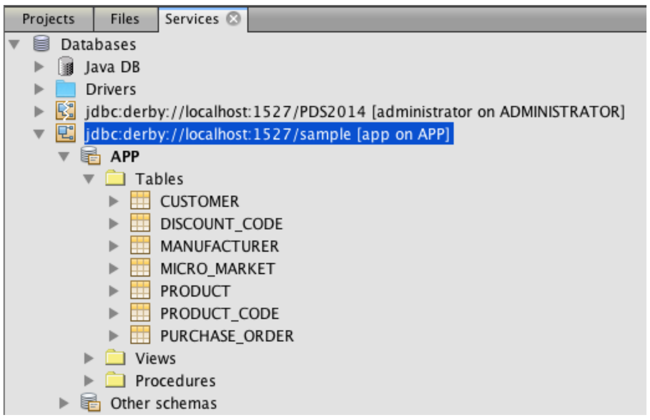


Figure 10: Admin Console Website

1.3 Databases

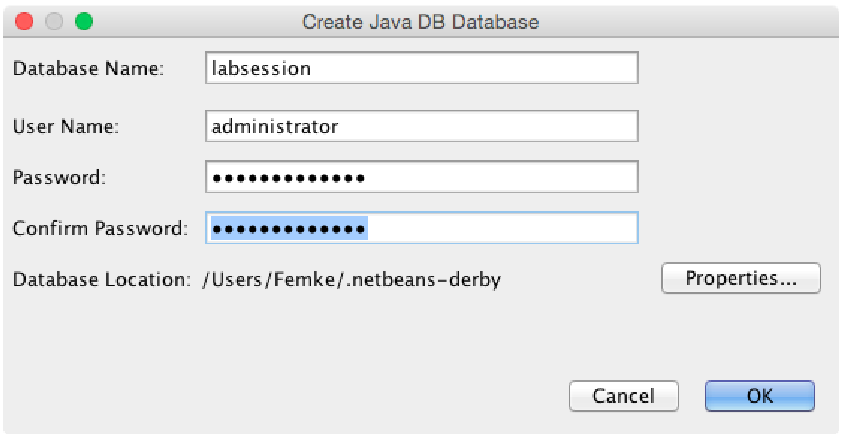
The Java DB is Sun's supported distribution of the open source Apache Derby database. Figure 11 shows the databases and database connections and the sample database.



```
Databases
  Java DB
  Drivers
  jdbc:derby://localhost:1527/PDS2014 [administrator on ADMINISTRATOR]
  jdbc:derby://localhost:1527/sample [app on APP]
    APP
      Tables
        CUSTOMER
        DISCOUNT_CODE
        MANUFACTURER
        MICRO_MARKET
        PRODUCT
        PRODUCT_CODE
        PURCHASE_ORDER
      Views
      Procedures
    Other schemas
```

Figure 11: Database view in the Services window

To create a database, click right on Java DB and choose Create Database.


Create Java DB Database

Database Name: labsession

User Name: administrator

Password: •••••••••••••

Confirm Password: •••••••••••••

Database Location: /Users/Femke/.netbeans-derby

Figure 12: Creation of database

Fill in the database name, and a username and password. Using SQL reserved keywords as username is not allowed.

2 Enterprise Beans

An Enterprise Bean is a server-side component that encapsulates the business logic of an application. For example: the enterprise beans might implement the business logic in an enterprise banking application in methods such as checkBalance, withdraw,. . .
Enterprise Beans run in the Enterprise Java Beans (EJB) container, a runtime environment within the Application Server. The EJB container provides system-level services to enterprise beans such as transaction management, security (authorization), lifecycle management, database mappings and transactions. Enterprise Beans simplify development of large, distributed applications, because the developer can concentrate on the business problems. The beans rather than the clients contain the application's business logic. Enterprise beans are portable components. There are two different types of Enterprise Beans: Session Beans or Message Driven Beans.

2.1 Session Beans

2.1.1 What is a Session Bean?

A **Session Bean** is associated with a single client inside the Application Server. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding the client from the complexity by executing business tasks inside the server.

A session bean is similar to an interactive session (one user). A session bean is not shared, can have only one client and is not persistent. When the client terminates, the session bean is no longer associated with the client.

2.1.2 Stateless, Stateful or Singleton Session Bean

Three types of session beans:

- **Stateful Session Bean**: The state of an object consists of the values of its instance variables. In a stateful session bean, the instance variables represent the state of a unique client-bean session. (This state is often called conversational state). This is an interactive session.
- **Stateless Session Bean**: A Stateless Session Bean does not maintain a conversational state with the client. When a client invokes the methods of a Stateless Bean, the bean's instance variables may contain a state specific to that client, but only for the duration of the invocation. When the method is finished, the client-specific state is not necessarily retained.
- **Singleton Session Bean**: A Singleton Session Bean is instantiated once per application, and exists for the lifecycle of the application. Singleton Session Beans are designed for circumstances where a single

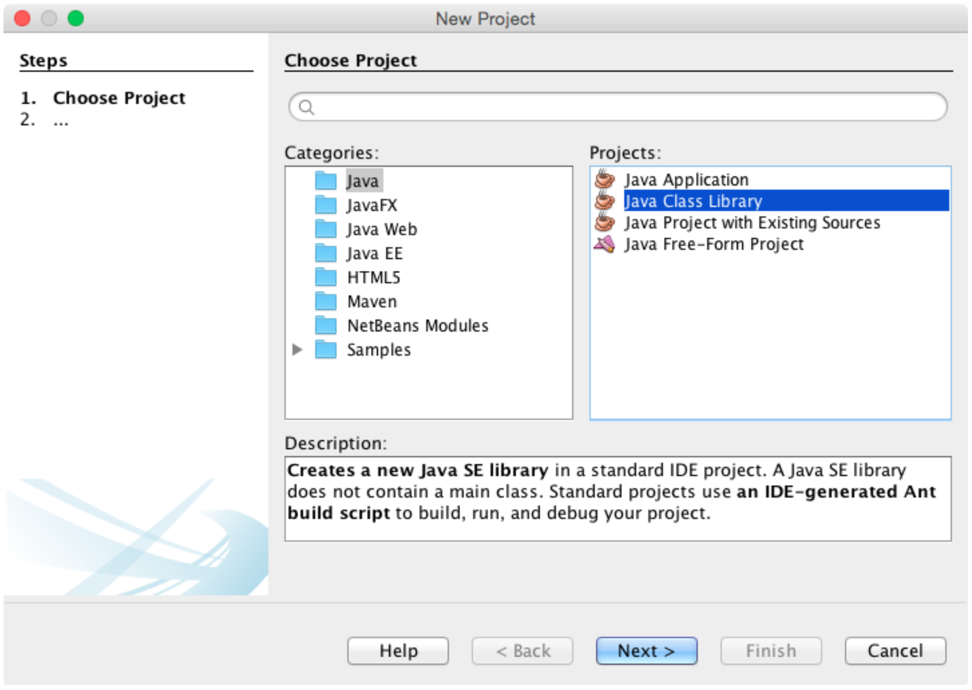
enterprise bean instance is shared across and concurrently accessed by clients. Singleton Session Beans offer similar functionality to stateless session beans, but differ from stateless session beans in that there is only one Singleton Session Bean per application, as opposed to a pool of Stateless Session Beans, any of which may respond to a client request.

Once you have created an Enterprise Application project, you are able to create a session bean inside the EJB subproject. This is illustrated in the Calculator Session Bean.

Calculator Session Bean

In this exercise we will create a calculator. The calculation is executed on the server-side. Sometimes complex calculations require more processing or memory, and are therefore executed server-side. In this simple example, we show a calculator that adds two numbers or multiplies two numbers.

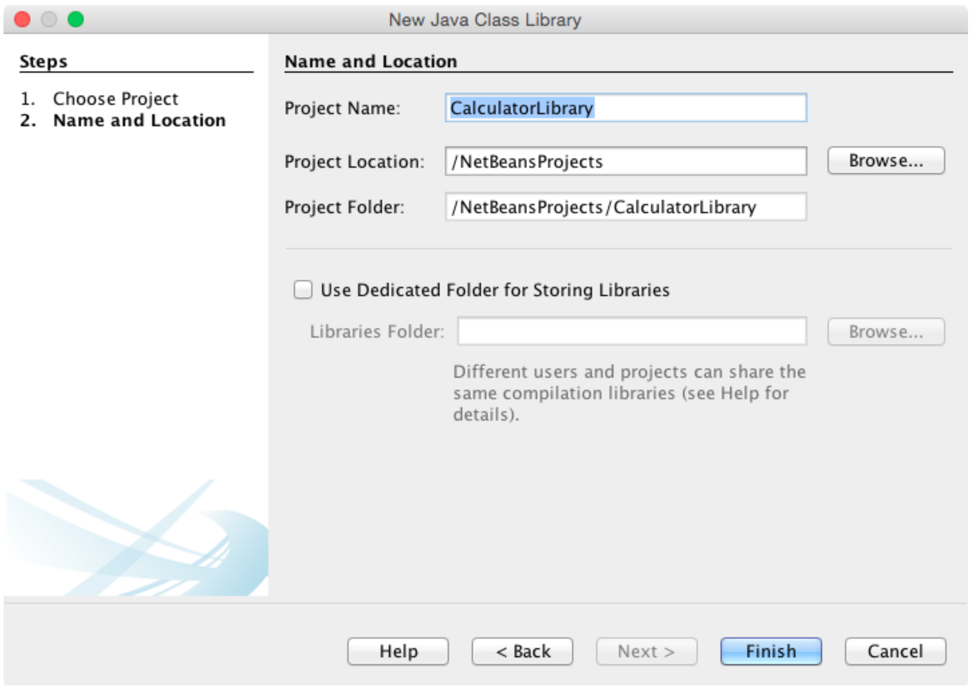
1. First create a Java Class Library to store the remote interfaces and util classes. This library will be used by the Enterprise Client Application and indicates which functionality is present within the back-end.



Creates a new Java SE library in a standard IDE project. A Java SE library does not contain a main class. Standard projects use **an IDE-generated Ant build script** to build, run, and debug your project.

Figure 13: Create Java Class Library

2. Call it `CalculatorLibrary` and click Finish.


New Java Class Library

Steps
1. Choose Project
2. **Name and Location**

Name and Location

Project Name: CalculatorLibrary

Project Location: /NetBeansProjects

Project Folder: /NetBeansProjects/CalculatorLibrary

☐ Use Dedicated Folder for Storing Libraries

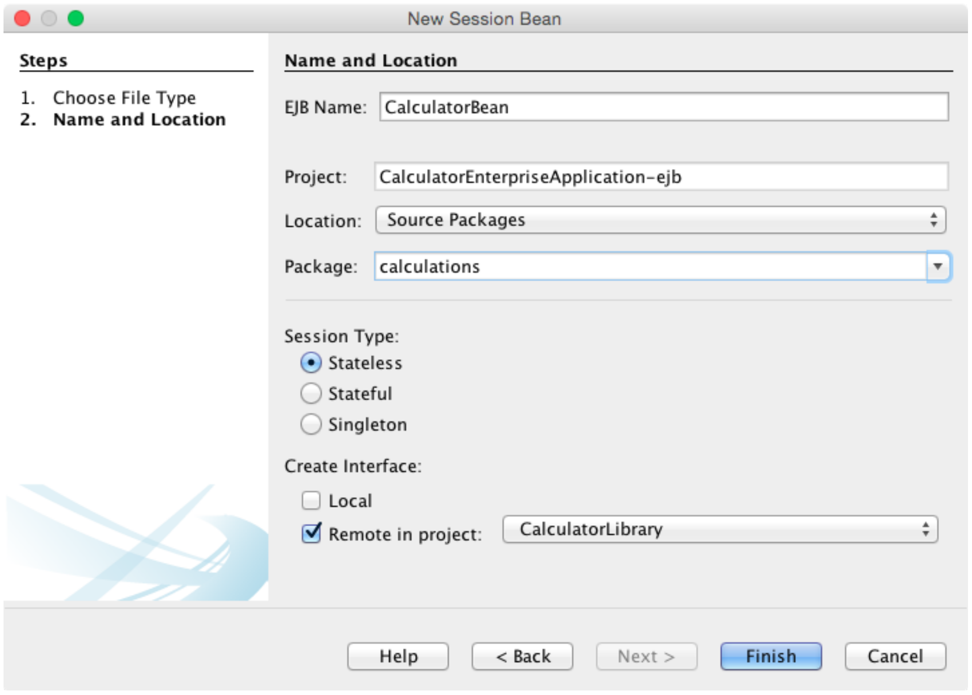
Libraries Folder:

Different users and projects can share the same compilation libraries (see Help for details).

Figure 14: Create Java Class Library, step 1

3. Now create the bean: right-click on the EJB project folder and choose New > Session Bean. The skeleton code for the session bean will be generated by Netbeans. Call your Stateless Session Bean `CalculatorBean`. Make sure you have selected the Remote option. Otherwise, the Client will not be able to communicate with the Bean.

   (In this exercise we will a Stateless session bean and we will only use a Remote interface which will be located in `CalculatorLibrary`).


New Session Bean

Steps
1. Choose File Type
2. **Name and Location**

Name and Location

EJB Name: CalculatorBean

Project: CalculatorEnterpriseApplication-ejb

Location: Source Packages

Package: calculations

Session Type:
- (●) Stateless
- ( ) Stateful
- ( ) Singleton

Create Interface:
- ☐ Local
- ☑ Remote in project: CalculatorLibrary

Figure 15: Create a Session Bean

4. Right-click on an empty line in the calculator Session Bean, choose Insert Code and select Add Business Method. Then fill in the method name, return type and parameters. First create the `sum` method. Click Add to add parameters to the method.

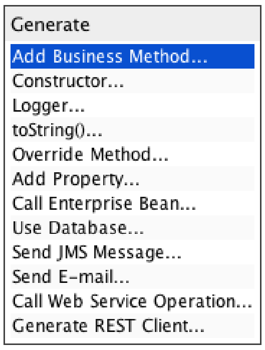
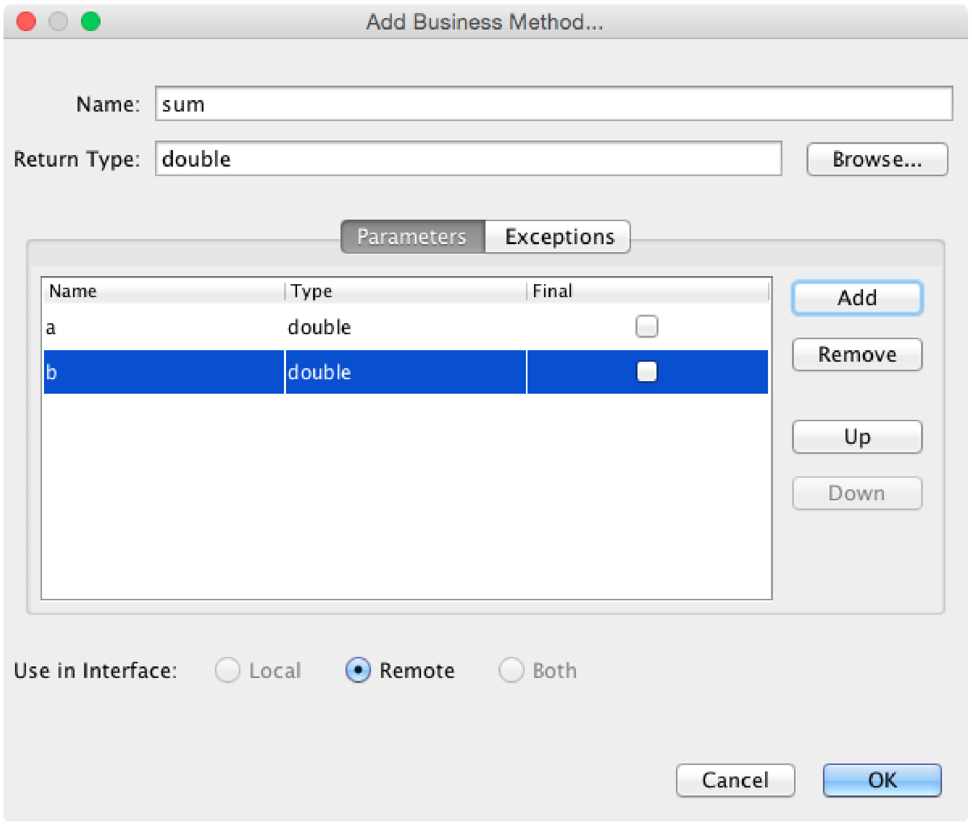


Figure 16: Insert Code



Name: sum

Return Type: double

Name	Type	Final
a	double	☐
b	double	☐

Use in Interface: Local [✓ Remote] Both

Figure 17: Add Business Method

5. Repeat step 2 and create the `product` method. Another possibility is to write the code, always make sure that your declare your method in the Remote interface.

   This code is generated after step 5. Change the implementation of the sum and product methods.

```java
package calculations;

import javax.ejb.Stateless;

@Stateless
public class CalculatorBean implements CalculatorBeanRemote {

    @Override
    public double sum(double a, double b) {
        return a + b;
    }

    @Override
    public double product(double a, double b) {
        return a * b;
    }

}
```

Following interface is automatically created in the `CalculatorLibrary` project:

```java
package calculations;

import javax.ejb.Remote;

@Remote
public interface CalculatorBeanRemote {

    double sum(double a, double b);

    double product(double a, double b);

}
```

6. Now, the application client will be created. Right-click in the main method of the application client and choose Insert Code, Call Enterprise Bean. Then choose the `CalculatorBean`. The code with the reference to the Bean is added.

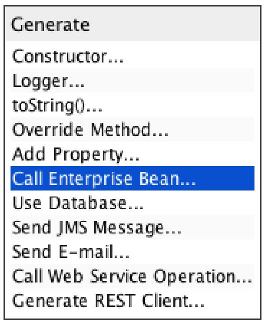
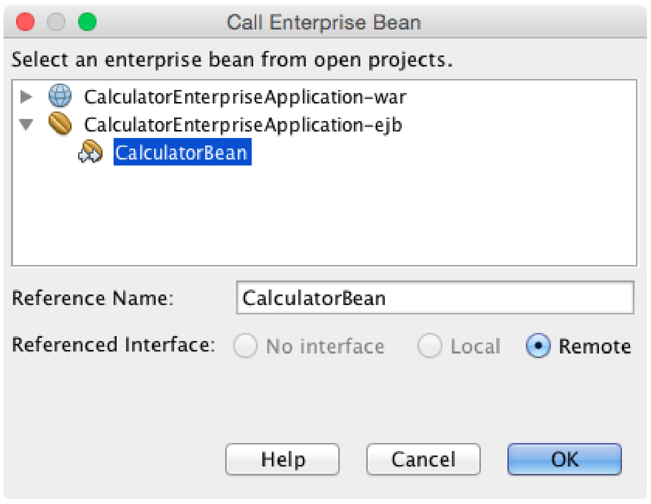


Figure 18: Call Enterprise Bean



Select an enterprise bean from open projects.

- CalculatorEnterpriseApplication-war
- CalculatorEnterpriseApplication-ejb
  - CalculatorBean

Reference Name: CalculatorBean

Referenced Interface: No interface [ ] Local [ ] [✓ Remote]

Figure 19: Call Enterprise Bean

```
    @EJB
    private static CalculatorBeanRemote calculatorBean;
```

(The generated client class in the application client is called Main. Here, we renamed (Refactor > Rename) it into Client.)

```
public class Client {

    @EJB
    private static CalculatorBeanRemote calculatorBean;

    public static void main(String[] args) {
        double a = 25.0;
        double b = 50.0;
        double sum = calculatorBean.sum(a, b);
        double product = calculatorBean.product(a, b);
        System.out.println("Sum:" + sum);
        System.out.println("Product:" + product);
    }

}
```

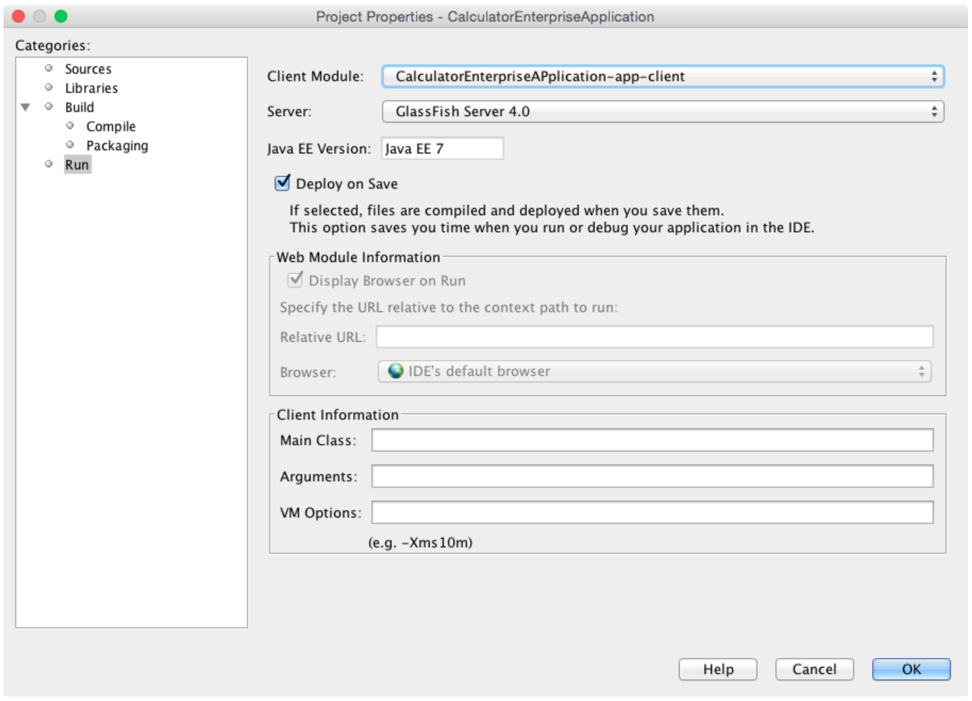
7. To run your application first deploy the enterprise application project.

   (a) Right-click on the enterprise and choose 'Clean and Build'

   (b) Right-click on the enterprise and choose 'Deploy'

   (c) Right-click on the enterprise and choose 'Run'

   (If the web application is started instead of the app-client, you can change this in Properties (Right-click). These settings can be found under Run. Figure 20 shows this configuration window.)


Project Properties - CalculatorEnterpriseApplication

Categories:
- Sources
- Libraries
- Build
  - Compile
  - Packaging
- Run

Client Module: CalculatorEnterpriseAPplication-app-client

Server: GlassFish Server 4.0

Java EE Version: Java EE 7

☑ Deploy on Save

If selected, files are compiled and deployed when you save them.
This option saves you time when you run or debug your application in the IDE.

Web Module Information

☑ Display Browser on Run

Specify the URL relative to the context path to run:

Relative URL:

Browser: IDE's default browser

Client Information

Main Class:

Arguments:

VM Options:

(e.g. -Xms10m)

Figure 20: Changing the run Properties

2.2 FriendList application: using the Java Persistence API

In this section we will show you how to create a simple **FriendList application** using the Netbeans IDE.

The FriendList application is a Java EE application that allows users to create and manage an account via a web interface. Users can be added to a list of friends or can be removed from the list. An account is characterized through a name and password. Each account holds a collection of friend-accounts.

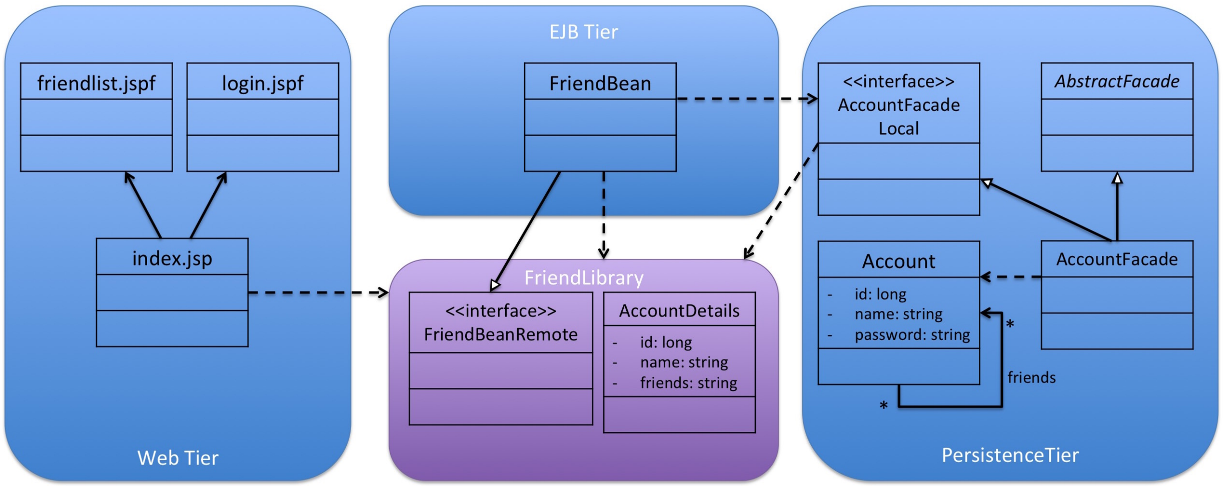


Figure 21: UML diagram showing the structure of the FriendList application

Suppose you have created a new empty Java Enterprise Application Project (with an `ejb-project` and `war-project`), follow the next steps:

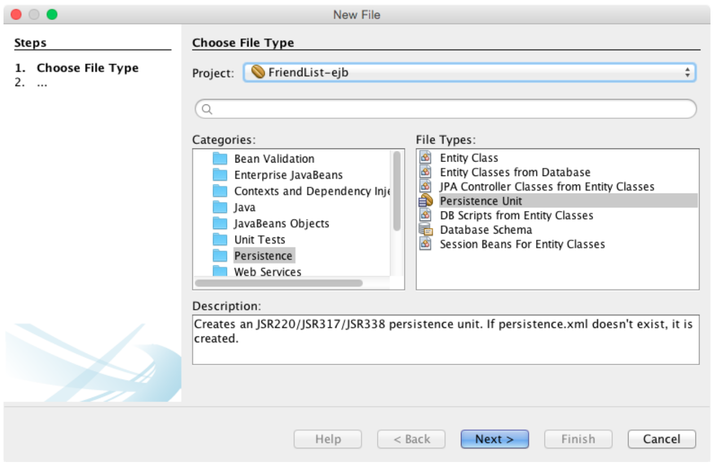
2.2.1 The Persistence Tier

The Java Persistence Unit and Entities

The **Java Persistence API** provides an object/relational mapping facility to Java developers for managing relational data in Java Applications. The Java Persistence API consists of the Java Persistence API, the query language, object/relational mapping metadata.

Before you can create an Entity Class, a Persistence Unit must be created. This persistence unit is the interface to the underlying database. It translates the Entities into database tables and method calls into database queries. Once a persistence unit and data source have been created, the entity classes can be written.

1. A database `friendlist` should be created first (in the Services window). Create the database and connect with this database.
2. If there is no Persistence Unit defined, click New File in the `ejb-project` and choose Persistence Unit (or select it through Other).
3. Now your Database should be linked to the Enterprise Application. Choose Data Source and create a new Data source with:
   - JNDI Name: `jdbc/friendlistdb`
   - Database Connection: `jdbc:derby://localhost:1527/friendlist` (or create a new database connection).
   - Remark: writing `jdbc/friendlistdb` in the Data Source field will not create a connection!



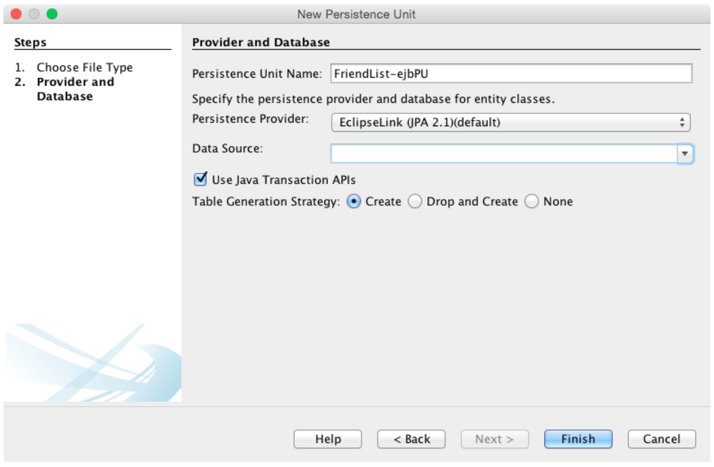
Project: FriendList-ejb

Categories:
- Bean Validation
- Enterprise JavaBeans
- Contexts and Dependency Inje
- Java
- JavaBeans Objects
- Unit Tests
- Persistence
- Web Services

File Types:
- Entity Class
- Entity Classes from Database
- JPA Controller Classes from Entity Classes
- Persistence Unit
- DB Scripts from Entity Classes
- Database Schema
- Session Beans For Entity Classes

Description:
Creates an JSR220/JSR317/JSR338 persistence unit. If persistence.xml doesn't exist, it is created.

Figure 22: Creation of Persistence Unit



Persistence Unit Name: FriendList-ejbPU

Specify the persistence provider and database for entity classes.

Persistence Provider: EclipseLink (JPA 2.1)(default)

Data Source:

☑ Use Java Transaction APIs

Table Generation Strategy: [✓ Create] Drop and Create None

Figure 23: Creation of Persistence Unit, step 2

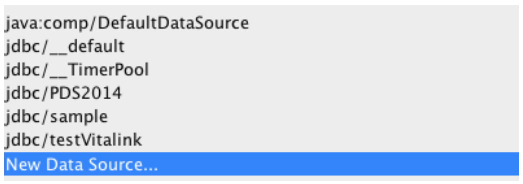
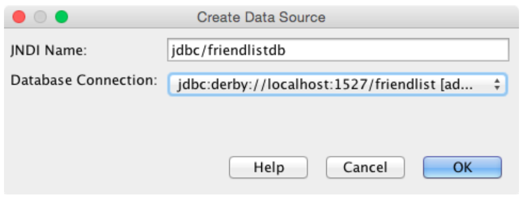

java:comp/DefaultDataSource
jdbc/__default
jdbc/__TimerPool
jdbc/PDS2014
jdbc/sample
jdbc/testVitalink
New Data Source...

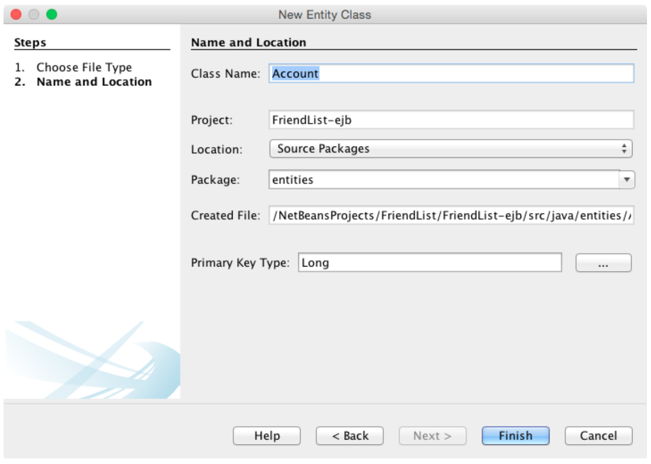
Figure 24: Creation of Data Source



JNDI Name: jdbc/friendlistdb
Database Connection: jdbc:derby://localhost:1527/friendlist [ad...

Figure 25: Creation of Data Source, step 2

4. Right-click on Source Packages in the EJB project (for example `FriendList-ejb`) and choose new Java Package and name it `entities`. Right-click on the <`entities`> package in the project explorer and choose New > Entity Class. You can also use the New File icon on the toolbar and choose Persistence > Entity Class in the dialog.



Class Name: Account
Project: FriendList-ejb
Location: Source Packages
Package: entities
Created File: /NetBeansProjects/FriendList/FriendList-ejb/src/java/entities/
Primary Key Type: Long

Figure 26: Creation of new Entity Class

5. Fill in the fields in the dialog, choose `Account` as class name and `entities` as package, as shown in the figure below.

   After you click the finish button, NetBeans will open a new entity class `Account` that contains the code:

```java
package entities;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Account implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof Account)) {
            return false;
        }
        Account other = (Account) object;
        if ((this.id == null && other.id != null) || (this.id != null && !this.id.equals(other.
            id))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "beans.Account[id=" + id + "]";
    }

}
```

Netbeans automatically generated some skeleton code for the newly created entity class. Note the `@Entity` annotation just above the class name, which denotes this class is an entity class. For now, the entity only has a single attribute, named `id`. Each of the entity's attributes will be translated into a column of the entity's database table. Using annotations, you can specify additional properties of the attributes. The annotations have to be put above the attribute definition or above its getter-method. The annotation `@GeneratedValue` specifies that the `id` attribute should be automatically created. When creating an entity instance it should thus not be set manually. This code is normally auto-generated and should not be changed.

6. Add the name, password and role properties to this entity class. This can be done by right-clicking on an empty line in the entity class and select Insert Code (Alt + Insert), Add Property to generate each property or you can just implement it in the file. Add the properties `String name`, `String password` and `String userRole`. The userRole property could be used in the GlassFish security realm, but this is outside the scope of this tutorial.

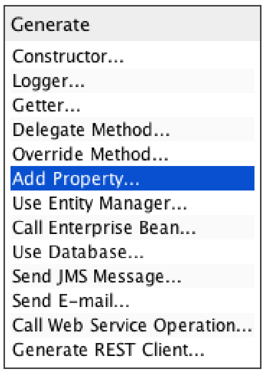
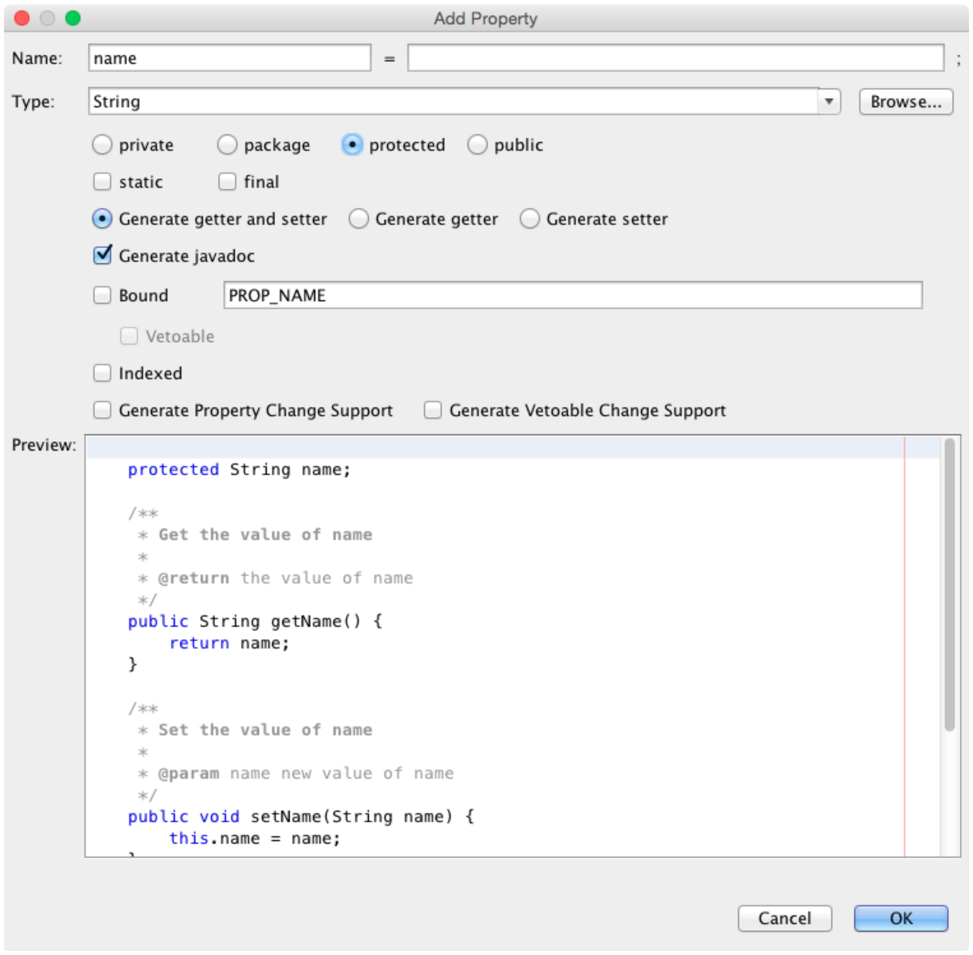


Figure 27: Insert Code - Add Property


Add Property

Name: name = ;

Type: String Browse...

( ) private ( ) package (●) protected ( ) public

[ ] static [ ] final

(●) Generate getter and setter ( ) Generate getter ( ) Generate setter

[✓] Generate javadoc

[ ] Bound PROP_NAME

[ ] Vetoable

[ ] Indexed

[ ] Generate Property Change Support [ ] Generate Vetoable Change Support

Preview:

```
protected String name;

/**
 * Get the value of name
 *
 * @return the value of name
 */
public String getName() {
    return name;
}

/**
 * Set the value of name
 *
 * @param name new value of name
 */
public void setName(String name) {
    this.name = name;
```

Cancel OK

Figure 28: Add Property

7. A regular column can be defined using the `@Column` annotation. Though, this is not absolutely necessary. The annotation can be used to set additional properties, such as whether this attribute is allowed to be set to null, or if it has to be unique. In this FriendList application, the account's name must be unique (i.e. `@Column(nullable = false, unique = true)`), while the password must not be unique (i.e. `@Column(nullable = false, unique = false)`). Add these annotations above the name and password variable. (An import to `javax.persistence.Column` will be requested).

8. Besides regular columns, annotations can also be used to define relationships between entities. The FriendList example contains one such relationship between different instances of the `Account` entity. Every `Account` has a list of other `Accounts`, which are its friends. As its friends can also have multiple other friends, this is a many-to-many relationship. Other possible types are one-to-one, one-to-many and many-to-one. Add a property friends to hold the `friends` collection. Select Insert Code > Add Property. Click the Browse button, type `Collection` and choose as type `java.util.Collection`.

9. Then change in the class the `Collection` into `Collection<Account>`. Info appears on the left side to define the relationship. Notice that a ManyToMany annotation is added. Select `2: Class Account` as this is the non-owning side of the relationship.

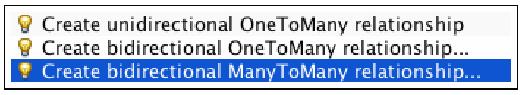
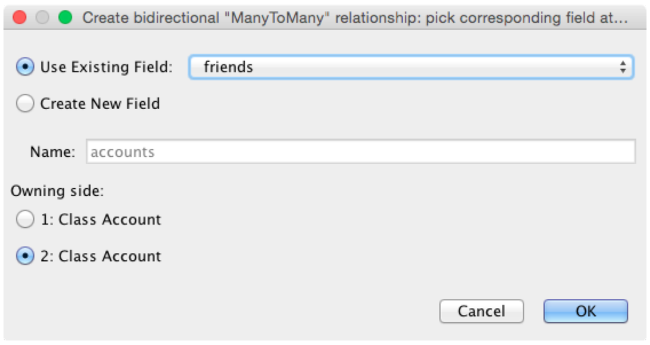

Create unidirectional OneToMany relationship
Create bidirectional OneToMany relationship...
Create bidirectional ManyToMany relationship...

Figure 29: Adding a relationship to the property `friends`



Create bidirectional "ManyToMany" relationship: pick corresponding field at...

(●) Use Existing Field: friends

( ) Create New Field

Name: accounts

Owning side:

( ) 1: Class Account

(●) 2: Class Account

Figure 30: Creation of the relationship ManyToMany

10. Add a constructor `public Account(String name, String password)` to the class and a default constructor `public Account()`. A default constructor in the entity class is always required!

```java
    public Account(String name, String password) {
        this.name = name;
        this.password = password;
        this.userRole = "user";
    }

    public Account() {
    }
```

11. We also want to define the methods within this class:

    - `boolean addFriend(Account friend)`
    - `boolean removeFriend(Account friend)`
    - `boolean hasFriend(Account friend)`

```java
    public boolean addFriend(Account friend) {
        if(friend == null || friends.contains(friend) || friend.equals(this))
            return false;
        else
            return friends.add(friend);
    }

    public boolean removeFriend(Account friend) {
        return friends.remove(friend);
    }

    public boolean hasFriend(Account friend) {
        return friends.contains(friend);
    }
```

12. An entity can contain one or more named queries. These can be used by facades to find entities based on certain properties. The `Account` entity defines two such queries. These respectively return the account with the specified name and all accounts that have a certain account as a friend. The code is inserted just below the `@Entity` annotation. Insert the named queries in the entity class. The named queries look like the code below. An import for `javax.persistence.NamedQueries` and for `javax.persistence.NamedQuery` will be suggested.

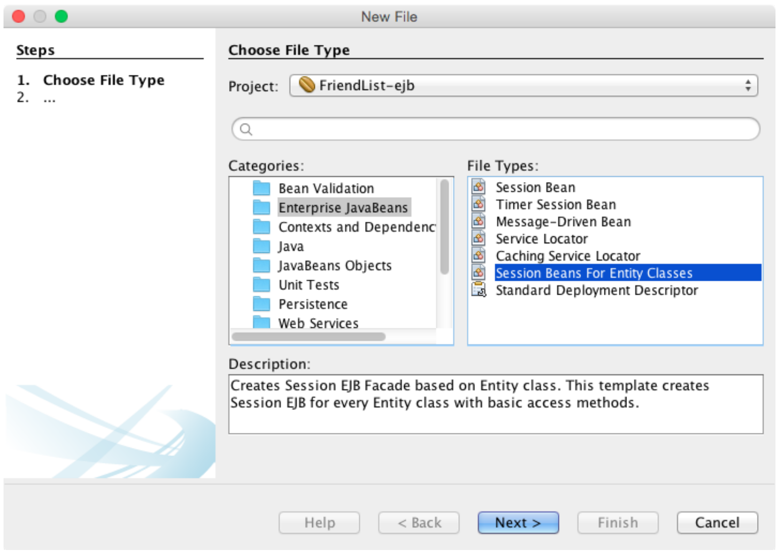
```
@NamedQueries({
    @NamedQuery(name = "entities.Account.findAccountByName",
    query = "SELECT p FROM Account p WHERE p.name = :name"),
    @NamedQuery(name = "entities.Account.findAccountByFriend", query =
    "SELECT DISTINCT p FROM Account p, IN(p.friends) f WHERE f = :friend")
})
```

Parameters in the named query are specified using a colon (e.g. `:name`). This parameter can be substituted by an actual value when the named query is called from the facade.

The Facades: Session Beans for the Persistence Tier

Facades provide a means to separate the business logic (EJB tier) of the Persistence Tier. It acts as a layer of abstraction on top of the entities. In this way, the business logic session beans, servlets and JSPs do not need to handle the entity classes themselves. This is done behind the scenes by the Facades. These Facades are created as stateless session beans.

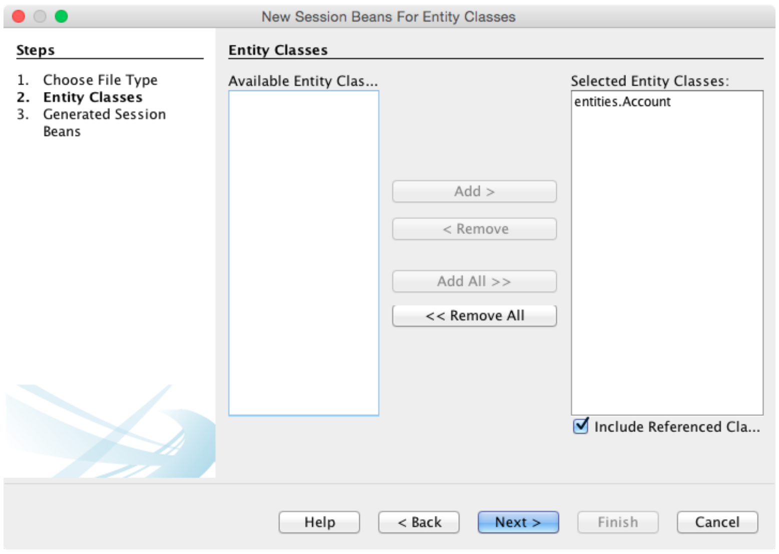
1. First, create a new package, named `facades`. Right-click on the <`facades`> package in the project explorer and choose New > Session Beans for Entity Classes. You can also use the New File icon on the toolbar and choose Java EE > Session Beans for Entity Classes in the dialog, as shown in Figure 31.



Creates Session EJB Facade based on Entity class. This template creates Session EJB for every Entity class with basic access methods.

Figure 31: Creation of the facade Stateless Session Bean when using the New File wizard

2. Select the entity `Account`.
3. In next dialog, the package where to place these facades and the types of interfaces can be chosen. Usually, we want these facades to only be accessed by the EJB tier, and not by the web tier. Therefore only a local interface should be created.


New Session Beans For Entity Classes

Steps
1. Choose File Type
2. **Entity Classes**
3. Generated Session Beans

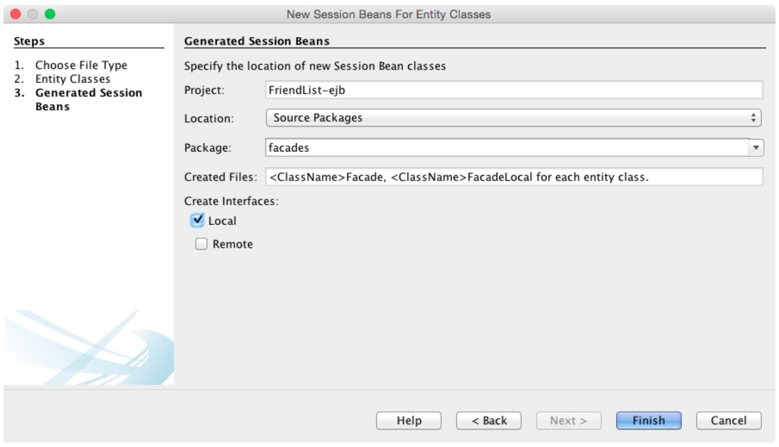
Entity Classes

Available Entity Clas...

Selected Entity Classes:
entities.Account

☑ Include Referenced Cla...

Figure 32: Selection of entity classes that need session beans


New Session Beans For Entity Classes

Steps
1. Choose File Type
2. Entity Classes
3. **Generated Session Beans**

Generated Session Beans

Specify the location of new Session Bean classes

Project: FriendList-ejb

Location: Source Packages

Package: facades

Created Files: <ClassName>Facade, <ClassName>FacadeLocal for each entity class.

Create Interfaces:
☑ Local
☐ Remote

Figure 33: Generating session beans

After you click Finish, the following classes are created:

```java
package facades;

import entities.Account;import java.util.List;
import javax.ejb.Local;

@Local
public interface AccountFacadeLocal {

    void create(Account account);

    void edit(Account account);

    void remove(Account account);

    Account find(Object id);

    List<Account> findAll();

    List<Account> findRange(int[] range);

    int count();
}
```

```java
package facades;

import java.util.List;
import javax.persistence.EntityManager;

public abstract class AbstractFacade<T> {
    private Class<T> entityClass;

    public AbstractFacade(Class<T> entityClass) {
        this.entityClass = entityClass;
    }

    protected abstract EntityManager getEntityManager();

    public void create(T entity) {
        getEntityManager().persist(entity);
    }

    public void edit(T entity) {
        getEntityManager().merge(entity);
    }

    public void remove(T entity) {
        getEntityManager().remove(getEntityManager().merge(entity));
    }

    public T find(Object id) {
        return getEntityManager().find(entityClass, id);
    }

    public List<T> findAll() {
        javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().
            createQuery();
        cq.select(cq.from(entityClass));
        return getEntityManager().createQuery(cq).getResultList();
    }

    public List<T> findRange(int[] range) {
        javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().
            createQuery();
        cq.select(cq.from(entityClass));
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        q.setMaxResults(range[1] - range[0] + 1);
        q.setFirstResult(range[0]);
        return q.getResultList();
    }

    public int count() {
        javax.persistence.criteria.CriteriaQuery cq = getEntityManager().getCriteriaBuilder().
            createQuery();
        javax.persistence.criteria.Root<T> rt = cq.from(entityClass);
        cq.select(getEntityManager().getCriteriaBuilder().count(rt));
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        return ((Long) q.getSingleResult()).intValue();
    }
```

```java
package facades;

import entities.Account;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class AccountFacade extends AbstractFacade<Account> implements AccountFacadeLocal {
    @PersistenceContext(unitName = "FriendList-ejbPU")
    private EntityManager em;

    @Override
    protected EntityManager getEntityManager() {
        return em;
    }

    public AccountFacade() {super(Account.class);}
}
```

The facade will now contain some basic methods for creating, editing, deleting and finding entities.

1. Add the following methods to the facade:

   - `private Account findByName(String name)`: In this methode, a named query is called to query the database for an `Account` with a specific name.
   - `private Account findById(long id)`

```java
    private Account findByName(String name) {
        List<Account> accounts = (List<Account>) em.createNamedQuery("entity.Account.
            findAccountByName")
                .setParameter("name", name).getResultList();
        if (accounts.size() == 1) {
            return accounts.get(0);
        } else if (accounts.size() > 1) {
            throw new IllegalStateException();
        } else {
            return null;
        }
    }

    private Account findById(long id) {
        return em.find(entity.Account.class, id);
    }
```

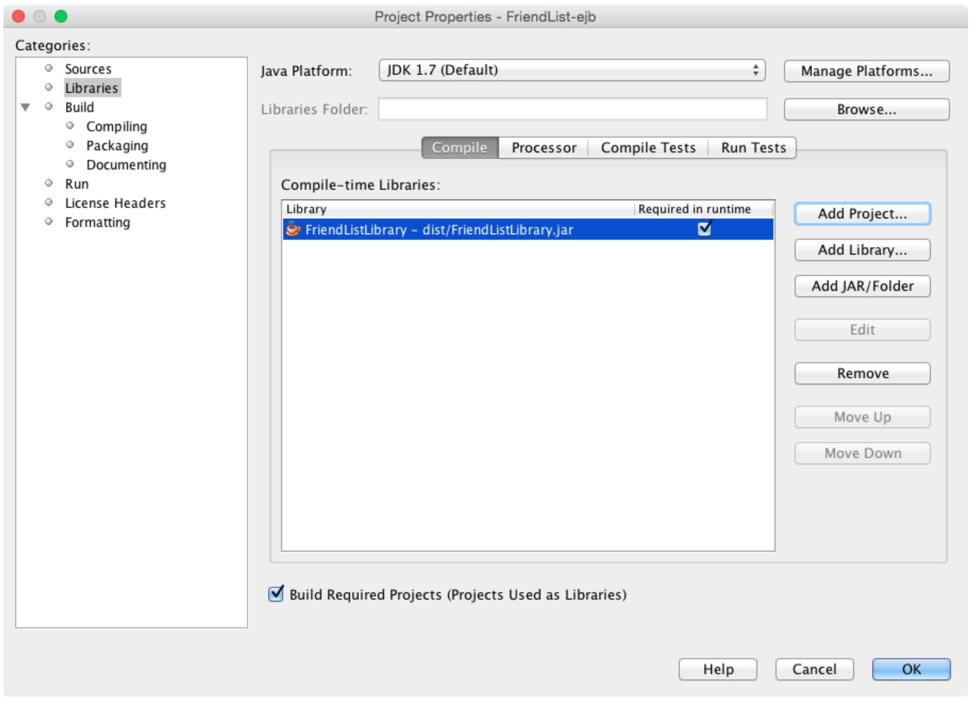
2. We also add other methods, we want to expose in the local interface. Add the methods to the `AccountFacadeLocal` interface:

   - `boolean createAccount(String name, String password);`
   - `boolean exists(String name);`
   - `Collection<String> getAllAccounts();`
   - `boolean addFriend(String name, String friend);`
   - `boolean removeFriend(String name, String friend);`
   - `boolean removeAccount(String name);`

```java
    public boolean createAccount(String name, String password) {
        if (exists(name)) {
            return false;
        }
        Account account = new Account(name, password);
        create(account);
        return true;
    }

    public boolean exists(String name) {
        return findByName(name) != null;
    }

    public Collection<String> getAllAccounts() {
        List<Account> accounts = (List<Account>) findAll();
        Collection<String> names = new ArrayList<String>();
        for (Account account : accounts) {
            names.add(account.getName());
        }
        return names;
    }

    public boolean addFriend(String name, String friend) {
        Account account = findByName(name);
        if (account != null) {
            Account fr = findByName(friend);
            if (fr != null) {
                boolean success = account.addFriend(fr);
                success = success && fr.addFriend(account);
                edit(account);
                edit(fr);
                return success;
            } else {
                return false;
            }
        } else {
            return false;
        }
    }

    public boolean removeFriend(String name, String friend) {
        Account account = findByName(name);
        if (account != null) {
            Account fr = findByName(friend);
            if (fr != null) {
                boolean success = account.removeFriend(fr);
                success = success && fr.removeFriend(account);
                edit(fr);
                edit(account);
                return success;
            } else {
                return false;
            }
        } else {
            return false;
        }
    }
```

```java
public boolean removeAccount(String name) {
    Account account = findByName(name);
    if (account != null) {
        List<Account> accounts = (List<Account>) em.createNamedQuery("entity.Account.
            findAccountByFriend")
                .setParameter("friend", account).getResultList();
        for (Account acc : accounts) {
            if (acc.hasFriend(account)) {
                acc.removeFriend(account);
                edit(acc);
            }
        }
        remove(account);
        return true;
    } else {
        return false;
    }
}
```

3. As with the calculator example, create a Java Class Library to contain the remote interfaces, call it `FriendListLibrary`.Make sure the `FriendListLibrary` is added to your EJB Project. Right-click on the project and select Properties.



Project Properties - FriendList-ejb

Categories: Sources, Libraries, Build (Compiling, Packaging, Documenting), Run, License Headers, Formatting

Java Platform: JDK 1.7 (Default)

Libraries Folder:

Compile-time Libraries:

Library	Required in runtime
FriendListLibrary - dist/FriendListLibrary.jar	✓

✓ Build Required Projects (Projects Used as Libraries)

Figure 34: Adding FriendLibrary to the Project

4. The last methods that should be added to the Facade are using the class `AccountDetails`. This class is used to prevent the application from directly accessing the entities. The facade returns the account information in the form of a wrapper class `util.AccountDetails` in the `FriendListLibrary` project.

   - `AccountDetails getAccount(String name);`
   - `AccountDetails getAccount(Long id);`

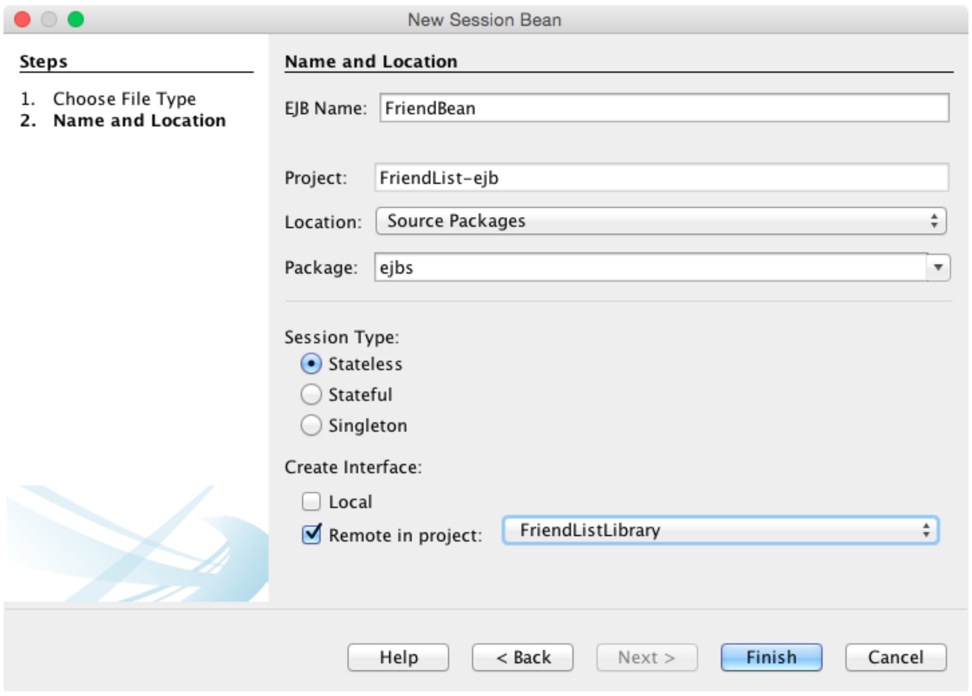
```java
    public AccountDetails getAccount(String name) {
        return convert(findByName(name), true);
    }

    public AccountDetails getAccount(Long id) {
        return convert(findById(id), true);
    }

    private AccountDetails convert(Account account,
            boolean includeFriendList) {
        if (account == null) {
            return null;
        } else {
            Collection<String> friends = new ArrayList<String>();
            if (includeFriendList) {
                for (Account friend : account.getFriends()) {
                    friends.add(friend.getName());
                }
            }
            return new AccountDetails(account.getName(), friends);
        }
    }
}
```

```java
package util;

import java.io.Serializable;
import java.util.Collection;

public class AccountDetails implements Serializable {

    private String name;
    private Collection<String> friends;

    public AccountDetails(String name, Collection<String> friends) {
        this.name = name;
        this.friends = friends;
    }

    public Collection<String> getFriends() {
        return friends;
    }

    public void setFriends(Collection friends) {
        this.friends = friends;
    }

    public String getName() {
        return name;
    }

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

2.2.2 The EJB Tier

The EJB tier contains the actual business logic. It communicates with the facades to store information in the database and provides interfaces with business logic methods which can be called from the web tier (or stand-alone application). It consists of session beans and message-driven beans. In this tutorial, only session beans will be discussed. For more information on both session and message-driven beans the reader is referred to the Java EE 7 tutorial.

1. Next right-click on Source Packages in the EJB project (for example `FriendList-ejb`) and choose new Java Package. Define a package named `ejb`. Right-click on the <`ejb`> package in the project explorer and choose New > Session Bean. You can also use the New File icon on the toolbar and choose Java EE > Session Bean in the dialog.



EJB Name: FriendBean
Project: FriendList-ejb
Location: Source Packages
Package: ejbs
Session Type:
- [✓ Stateless]
- Stateful
- Singleton

Create Interface:
- Local
- [✓ Remote in project:] FriendListLibrary

Figure 35: Creation of a session bean

2. Fill in the fields in the dialog. Choose `FriendBean` as class name and `ejbs` as package, as shown in the figure. The bean type and interface type can be selected. As our session beans need to be accessed from the web tier, a `remote` interface should be created. If the bean also needs to be accessed by other beans, a local interface can also be created. Select as session Type: Stateless and as interface: Remote and `FriendListLibrary` as destination project for the remote interfaces.

   There are three different bean types: Stateless, Stateful and Singleton. A Stateful Session Bean is unique for a single client and can keep track of that client's state. Such a client will not always be the end-user, but could be a servlet or JSP. As such a JSP instance can be shared by multiple users. Therefore, the Stateless Session Bean is generally a better choice when using a web container. The JSPs and servlets can keep track of user sessions and provide session information when calling a Session Bean method. A Stateless Session Bean is also used in this FriendList example.

3. Right-click in the `FriendBean` class and select Insert Code > Call Enterprise Bean. Select the `Account Facade` in the FriendList-ejb project. The Reference Name is `AccountFacade` and the Referenced Interface is Local.

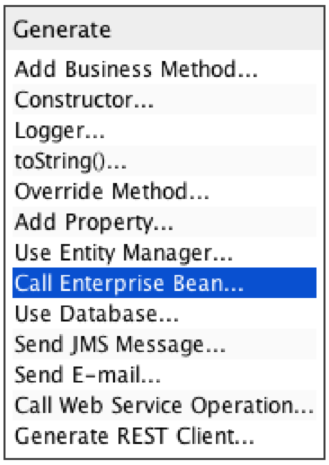


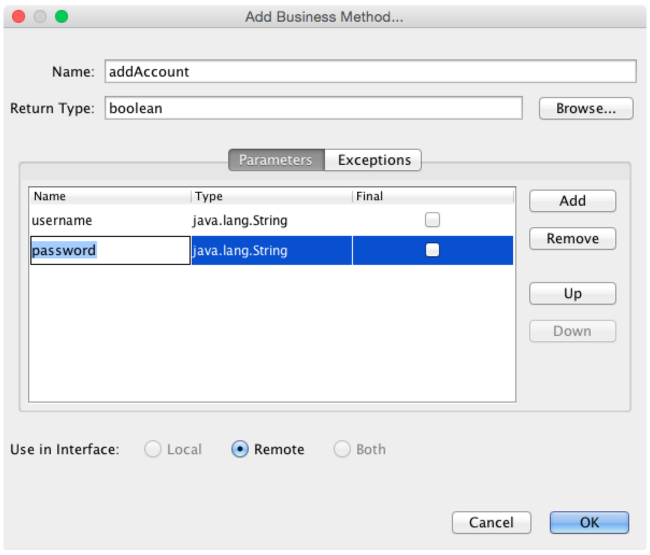
Figure 36: Call Enterprise Bean

The following code for the `FriendBean` and its remote interface `FriendBeanRemote` was generated:

```java
package ejbs;

import facades.AccountFacadeLocal;
import javax.ejb.EJB;
import javax.ejb.Stateless;

@Stateless
public class FriendBean implements FriendBeanRemote {

    @EJB
    private AccountFacadeLocal accountFacade;
```

```java
package ejbs;

import javax.ejb.Remote;

@Remote
public interface FriendBeanRemote {

}
```

4. Right-click in the `FriendBean` class and select Insert Code > Add Business Method. Now, we can define the different business methods.


Add Business Method...

Name: addAccount

Return Type: boolean

Name	Type	Final
username	java.lang.String	☐
password	java.lang.String	☐

Use in Interface: Local [✓ Remote] Both

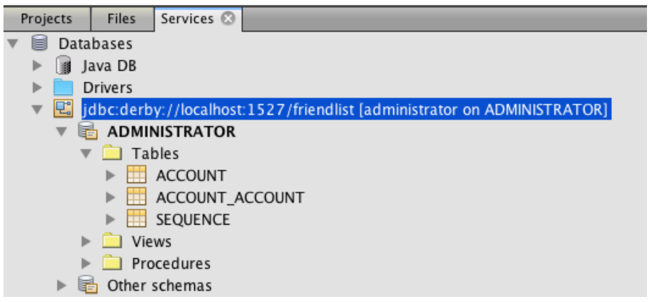
Figure 37: Add Business Method

We define the following methods:

- `AccountDetails getAccountInformation(String username)`
- `boolean addFriend(String username, String friend)`
- `boolean removeFriend(String username, String friend)`
- `boolean removeAccount(String username)`
- `boolean addAccount(String username, String password)`

```java
package ejbs;

import facades.AccountFacadeLocal;
import javax.ejb.EJB;
import javax.ejb.Stateless;
import util.AccountDetails;

@Stateless
public class FriendBean implements FriendBeanRemote {

    @EJB
    private AccountFacadeLocal accountFacade;

    public AccountDetails getAccountInformation(String username) {
        return accountFacade.getAccount(username);
    }

    public boolean addFriend(String username, String friend) {
        try {
            return accountFacade.addFriend(username, friend);
        } catch (Exception e) {
            return false;
        }
    }

    public boolean removeFriend(String username, String friend) {
        try {
            return accountFacade.removeFriend(username, friend);
        } catch (Exception e) {
            return false;
        }
    }

    public boolean removeAccount(String username) {
        try {
            return accountFacade.removeAccount(username);
        } catch (Exception e) {
            return false;
        }

    }

    public boolean addAccount(String username, String password) {
        try {
            if (!accountFacade.exists(username)) {
                return accountFacade.createAccount(username, password);
            } else {
                return false;
            }
        } catch (Exception e) {
            return false;
        }
    }
}
```

5. Clean and build the application. Deploy the FriendList application. Then check the database (in the Services window). New tables were created in the database. If not, check if you added the a default constructor in `Account.java`.



Projects | Files | Services

- Databases
  - Java DB
  - Drivers
  - jdbc:derby://localhost:1527/friendlist [administrator on ADMINISTRATOR]
    - ADMINISTRATOR
      - Tables
        - ACCOUNT
        - ACCOUNT_ACCOUNT
        - SEQUENCE
      - Views
      - Procedures
    - Other schemas

Figure 38: The created database tables in the Services window

2.2.3 The Web Tier

The web tier is used to provide a web-based user interface on top of the business logic. An alternative would be a stand-alone application. But as we will use a web-based interface for this exercise, only this is discussed in this tutorial.

Java Server Pages (JSP)

JSPs are HTML pages in which Java code can be inserted. Therefore they are best suited for visualization purposes. Servlets on the other hand are better suited for logic, with only little visualization content.

1. Open the `index.jsp` JSP webpage in the editor. The JSP page is located in the Web Pages folder of the FriendList-war project. If you want to create another JSP page right-click on Web Pages and select New > JSP. Look at the given code. The `FriendList` example contains one basic JSP page (`index.jsp`) and implements the `addAccount` functionality. Check out the created web page. Regular Java code is inserted using the tags < % %>. For example:

```

<%
    String username = (String) request.getParameter("username");
    String password = (String) request.getParameter("password");
%>
```

   Importing `javax.naming.*` and other used classes form the `FriendListLibrary` is necessary when calling an EJB. This is done in the following way:

```
<%@ page import="ejb.FriendBeanRemote, util.AccountDetails, javax.naming.*"%>

<%!
    private FriendBeanRemote friendRemote = null;

    public void jspInit() {
        try {
            InitialContext ic = new InitialContext();
            friendRemote = (FriendBeanRemote) ic.lookup(FriendBeanRemote.class.getName());
        } catch (Exception ex) {
            System.out.println("Couldn't create friendRemote bean." + ex.getMessage());
        }
    }

    public void jspDestroy() {
        friendRemote = null;
    }
%>
```

2. Paste the following code in `index.jsp`.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<%@ page import="ejb.FriendBeanRemote, util.AccountDetails, javax.naming.*"%>

<%!
    private FriendBeanRemote friendRemote = null;

    public void jspInit() {
        try {
            InitialContext ic = new InitialContext();
            friendRemote = (FriendBeanRemote) ic.lookup(FriendBeanRemote.class.getName());
        } catch (Exception ex) {
            System.out.println("Couldn't create friendRemote bean." + ex.getMessage());
        }
    }

    public void jspDestroy() {
        friendRemote = null;
    }
%>

<%
    String username = (String) request.getParameter("username");
    String password = (String) request.getParameter("password");
%>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>FriendList</title>
    </head>
    <body>

        <h2>Add account</h2>

        <%
            String error = null;

            if (username != null) {
                if (friendRemote.addAccount(username, password)) {
                    response.sendRedirect("index.jsp");
                } else {
                    error = "Adding account failed";
                }
            }

            if (error != null) {
        %>
                <p><font color="red"><% out.print(error);
                    }%></font></p>

        <form action="index.jsp" method="POST">
            <p>
                <input type="text" name="username" value="" size="20" />
                </br>
                <input type="password" name="password" value="" size="20" />
                </br>
                <input type="submit" value="Add Account" />
            </p>
        </form>
    </body>
</html>
```

3. Right-click on the `FriendList` project, click Clean and build, Deploy. Right-click on the `FriendList` project, click Run. Your browser will open `http://localhost:8080/FriendList-war/index.jsp`

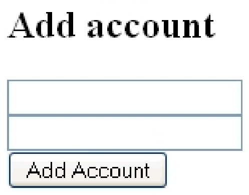


Figure 39: View of index webpage

Servlets

A servlet can be created by right clicking on the WAR project and selecting New > Servlet. You can then fill in the name of the class and package and also the servlet's name and the URL(s) that will point to this servlet. This will create the servlet, with some skeleton code that shows you how to create HTML output. Calling an EJB from a servlet is done in the same way as from another EJB. The difference is that this time we will want to call the bean's remote interface, instead of its local one.

3 Introduction to REST

REST stands for Representational State Transfer. (It is sometimes spelled “ReST”.) It relies on a stateless, client-server, cacheable communications protocol and in virtually all cases, the HTTP protocol is used. REST is an architecture style for designing networked applications. The idea is that, rather than relying on other mechanisms such as CORBA, RPC or SOAP to establish a connection between machines, simple HTTP is used to make calls between machines. In many ways, the World Wide Web itself, based on HTTP, can be viewed as a REST-based architecture.

RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations. REST is a lightweight alternative to mechanisms like RPC (Remote Procedure Calls) and Web Services (SOAP, WSDL,. . .). Despite being simple, REST is fully-featured; there is basically nothing you can do in Web Services that can’t be done with a RESTful architecture. However, REST is not a “standard”. There will never be a W3C recommendation for REST, for example.

3.1 REST as lightweight Web Services

As a programming approach, REST is a lightweight alternative to Web Services and RPC. Much like Web Services, a REST service is:

- Platform-independent (you do not care if the server is Unix, the client is a Mac, or anything else),
- Language-independent (C# can talk to Java, etc.)
- Standards-based (runs on top of HTTP), and
- Can easily be used in the presence of firewalls.

Like Web Services, REST offers no built-in security features, encryption, session management and QoS guarantees. But also as with Web Services, these can be added by building on top of HTTP:

- For security, username/password tokens are often used.
- For encryption, REST can be used on top of HTTPS (secure sockets).

One thing that is not part of a good REST design is cookies: The “ST” in “REST” stands for “State Transfer”, and indeed, in a good REST design operations are self-contained, and each request carries with it (transfers) all the information (state) that the server needs in order to complete it.

3.2 How is REST used?

We will take a simple web service as an example: querying a phonebook application for the details of a given user. All we have is the user’s ID. Using Web Services and SOAP, the request would look something like this:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:body pb="http://www.acme.com/phonebook">
      <pb:GetUserDetails>
        <pb:UserID>12345</pb:UserID>
      </pb:GetUserDetails>
  </soap:Body>
</soap:Envelope>
```

This message has to be sent (using an HTTP POST request) to the server. The result is often an XML file, but it will be embedded, as the ”payload”, inside a SOAP response envelope. When using REST the query will look like this:

```
http://www.acme.com/phonebook/UserDetails/12345
```

Note that this is not the request body, it’s just a URL. This URL is sent to the server using a simple GET request and the HTTP reply is the raw result data, not embedded. With REST, a simple network connection is the only requirement. You can even test the API directly, using your browser. Still, REST libraries (for simplifying things) do exist, for example RESTlet.

Also, note how the URLs "method" part is not called "GetUserDetails", but simply "UserDetails". It is a common convention in REST design to use nouns rather than verbs to denote simple resources.
REST can easily handle more complex requests, including multiple parameters.
For example, the following GET operation:

```
http://www.acme.com/phonebook/UserDetails?firstName=John&lastName=Doe
```

As a rule, GET requests should be for read-only queries; they should not change the state of the server and its data. For creation, updating, and deleting data, use POST, PUT and DELETE requests.

While REST services might use XML in their responses (as one way of organizing structured data), REST requests rarely use XML. As shown above, in most cases, request parameters are simple, and there is no need for the overhead of XML. One advantage of using XML is type safety. However, in a stateless system like REST, you should always verify the validity of your input, XML or otherwise. Another way to generate responses is using JSON (http://json.org). JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.

3.3 REST architecture components

The key components of a REST architecture are:

- Resources, which are identified by logical URLs. Both state and functionality are represented using resources.
  - The logical URLs imply that the resources are universally addressable by other parts of the system.
  - Resources are the key element of a true RESTful design, as opposed to "methods" or "services" used in RPC and SOAP Web Services, respectively. You do not issue a "getProductName" and then a "getProductPrice" RPC calls in REST; rather, you view the product data as a resource and this resource should contain all the required information (or links to it).
- A web of resources, meaning that a single resource should not be overwhelmingly large and contain too fine-grained details. Whenever relevant, a resource should contain links to additional information, just as in web pages.
- The system has a client-server, but of course one component's server can be another component's client.
- There is no connection state; interaction is stateless (although the servers and resources can of course be stateful). Each new request should carry all the information required to complete it, and must not rely on previous interactions with the same client.
- Resources should be cachable whenever possible (with an expiration date/time). The protocol must allow the server to explicitly specify which resources may be cached, and for how long.
  - Since HTTP is universally used as the REST protocol, the HTTP cache-control headers are used for this purpose.
  - Clients must respect the server's cache specification for each resource.
- Proxy servers can be used as part of the architecture, to improve performance and scalability. Any standard HTTP proxy can be used.
- Note that your application can use REST services (as a client) without being a REST architecture by itself; e.g., a single-machine, non-REST program can access 3rd-party REST services.

3.4 Creating RESTful Web Services from a Database in NetBeans

The NetBeans IDE supports rapid development of RESTful web services using JSR 311 - Java API for RESTful Web Services (JAX-RS) and Jersey, the reference implementation for JAX-RS. More information can be found in:

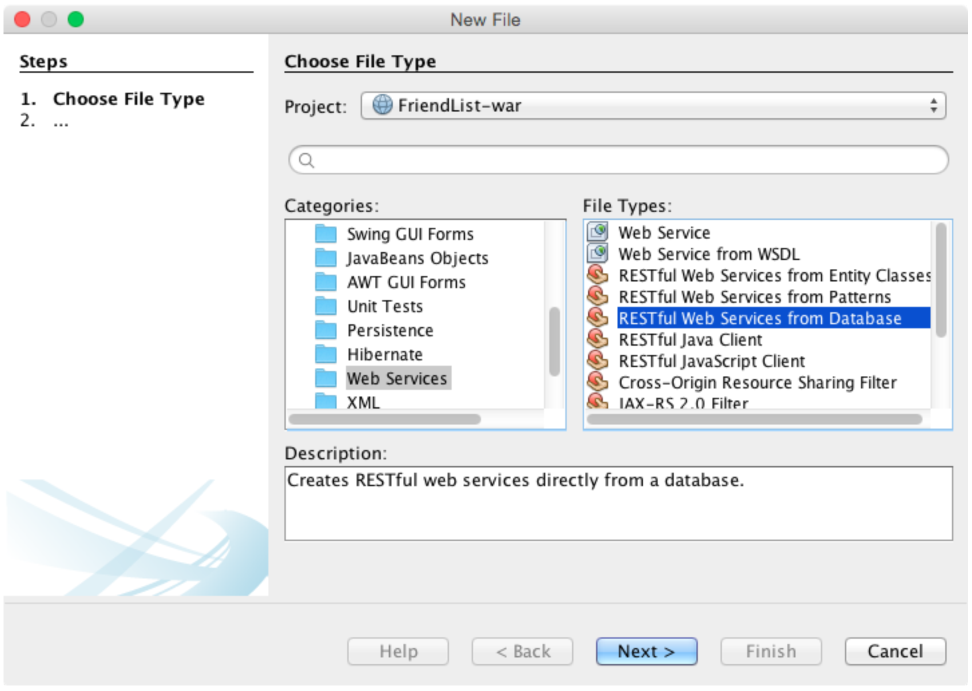
- JSR 311, the Java API for RESTful Web Services: `http://jcp.org/en/jsr/detail?id=311`
- Jersey, the reference implementation: `http://jersey.dev.java.net/`

In this tutorial, we will focus on building a RESTful Web Service from a database[1]. NetBeans provides other features for RESTful Web Services:

- Rapid creation of RESTful web services from JPA entity classes and patterns.
- Rapid code generation for invoking web services such as Google Map, Yahoo News Search, and StrikeIron web services by drag-and-dropping components from the Web Services manager in the Services window.
- Generation of RESTful Java Clients for services registered in the Web Services manager.
- Test client generation for testing RESTful web services.
- Logical view for easy navigation of RESTful web service implementation classes in your project.

We will use the tables, created in the FriendsList application, as starting point to create a RESTful Web Service. This will also show you how entities are generated, based on the structure and information from a database.

1. Right-click the FriendList-war node and choose New > Other > Web Services > RESTful Web Services from Database. The New RESTful Web Service wizard opens, on the Database Tables panel.



Project: FriendList-war

Description: Creates RESTful web services directly from a database.

Figure 40: Creating a new RESTful Web Service from Database

[1]https://netbeans.org/kb/docs/websvc/rest.html, other tutorials and blogs can be found at `https://netbeans.org/kb/trails/web.html`

2. Select the `Account` table. The `Account_Account` table will be selected automatically.

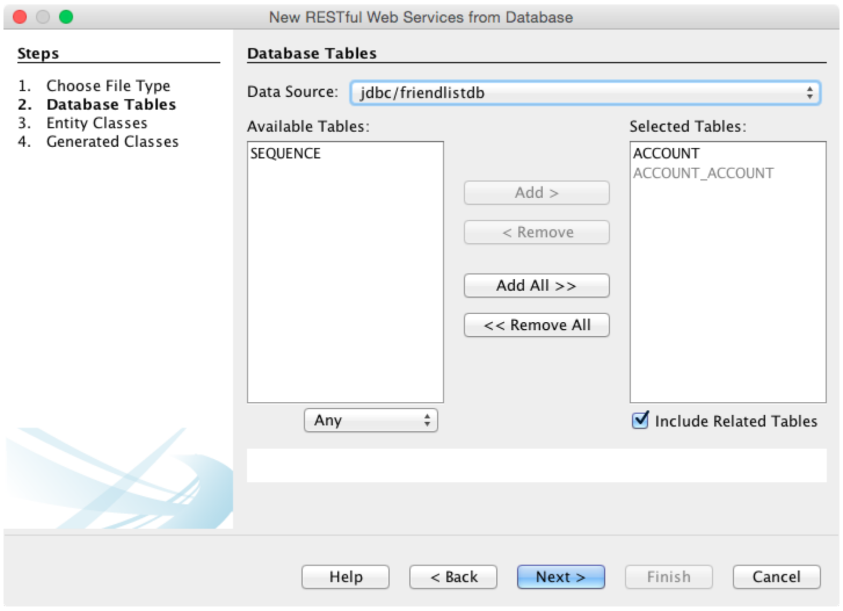
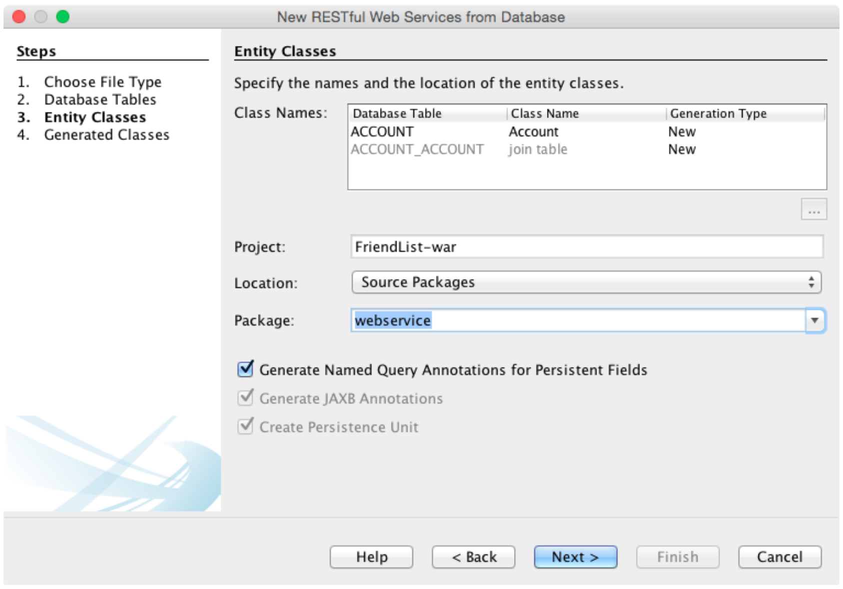


Figure 41: Creating a new RESTful Web Service from Database, step 2

3. Choose a package new, where the created classes can be located and click Finish.



Specify the names and the location of the entity classes.

Class Names:

Database Table	Class Name	Generation Type
ACCOUNT	Account	New
ACCOUNT_ACCOUNT	join table	New

Project: FriendList-war

Location: Source Packages

Package: webservice

☑ Generate Named Query Annotations for Persistent Fields

☑ Generate JAXB Annotations

☑ Create Persistence Unit

Figure 42: Creating a new RESTful Web Service from Database, step 3

4. Figure 43 lists the packages and classes that should be created in the `Source Packages`. When you look to these classes in detail, you will see that the generated `Account` entity looks a bit different from the one that was created earlier.

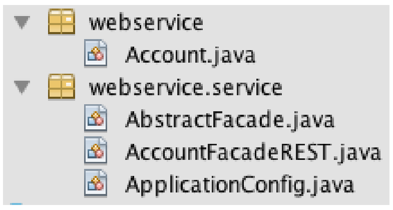


Figure 43: Created packages and classes

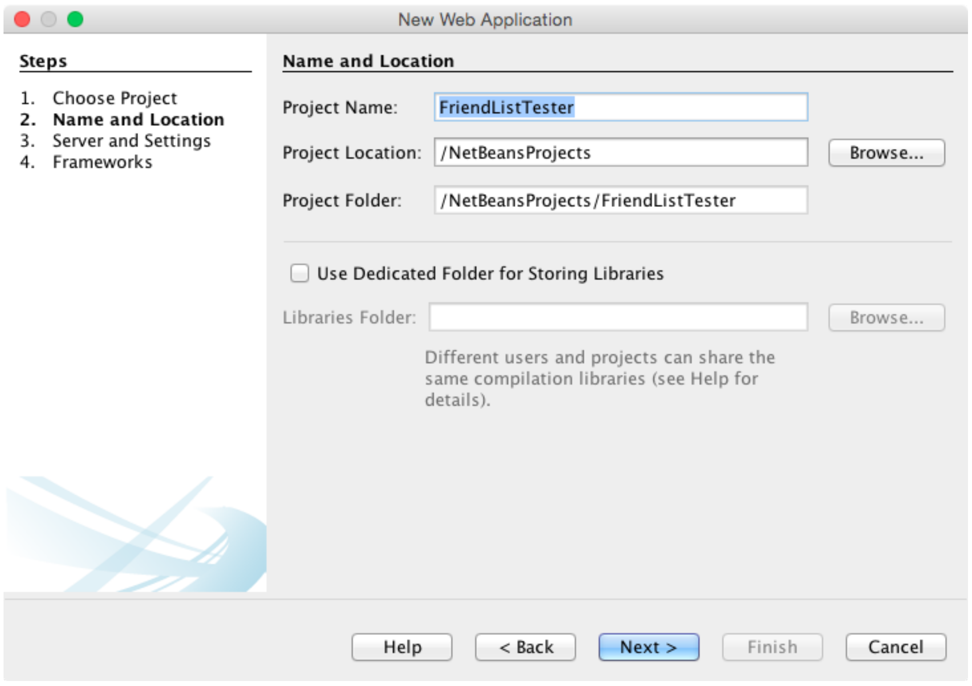
AccountFacadeREST has been enriched with annotations to indicate which methods match the different CRUD operations and which format is consumed and produced. The @Path indicates how the URL should be extended to request information.

```java
package webservice.service;

import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.*;
import javax.ws.rs.*;
import webservice.Account;

@Stateless
@Path("webservice.account")
public class AccountFacadeREST extends AbstractFacade<Account> {
    @PersistenceContext(unitName = "RESTWebServicePU")
    private EntityManager em;

    public AccountFacadeREST() {
        super(Account.class);
    }

    @POST
    @Override
    @Consumes({"application/xml", "application/json"})
    public void create(Account entity) {
        super.create(entity);
    }

    @PUT
    @Path("{id}")
    @Consumes({"application/xml", "application/json"})
    public void edit(@PathParam("id") Long id, Account entity) {
        super.edit(entity);
    }

    @DELETE
    @Path("{id}")
    public void remove(@PathParam("id") Long id) {
        super.remove(super.find(id));
    }

    @GET
    @Path("{id}")
    @Produces({"application/xml", "application/json"})
    public Account find(@PathParam("id") Long id) {
        return super.find(id);
    }

    @GET
    @Override
    @Produces({"application/xml", "application/json"})
    public List<Account> findAll() {
        return super.findAll();
    }

    @GET
    @Path("{from}/{to}")
    @Produces({"application/xml", "application/json"})
    public List<Account> findRange(@PathParam("from") Integer from, @PathParam("to") Integer to)
         {
        return super.findRange(new int[]{from, to});
    }
    @GET
    @Path("count")
    @Produces("text/plain")
    public String countREST() {
        return String.valueOf(super.count());
    }

    @Override
    protected EntityManager getEntityManager() {
        return em;
    }

}
```

`ApplicationConfig` sets the Application Path and gathers all the resources defined in the Facade.

```java
package webservice.service;

import java.util.Set;
import javax.ws.rs.core.Application;

@javax.ws.rs.ApplicationPath("webresources")
public class ApplicationConfig extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        addRestResourceClasses(resources);
        return resources;
    }

    /**
     * Do not modify addRestResourceClasses() method.
     * It is automatically populated with
     * all resources defined in the project.
     * If required, comment out calling this method in getClasses().
     */
    private void addRestResourceClasses(Set<Class<?>> resources) {
        resources.add(webservice.service.AccountFacadeREST.class);
    }

}
```

5. Next, create a new WAR project `FriendListTester`. In this project, we will test the RESTful Web Service.


New Web Application

Steps
1. Choose Project
2. **Name and Location**
3. Server and Settings
4. Frameworks

Name and Location

Project Name: FriendListTester

Project Location: /NetBeansProjects

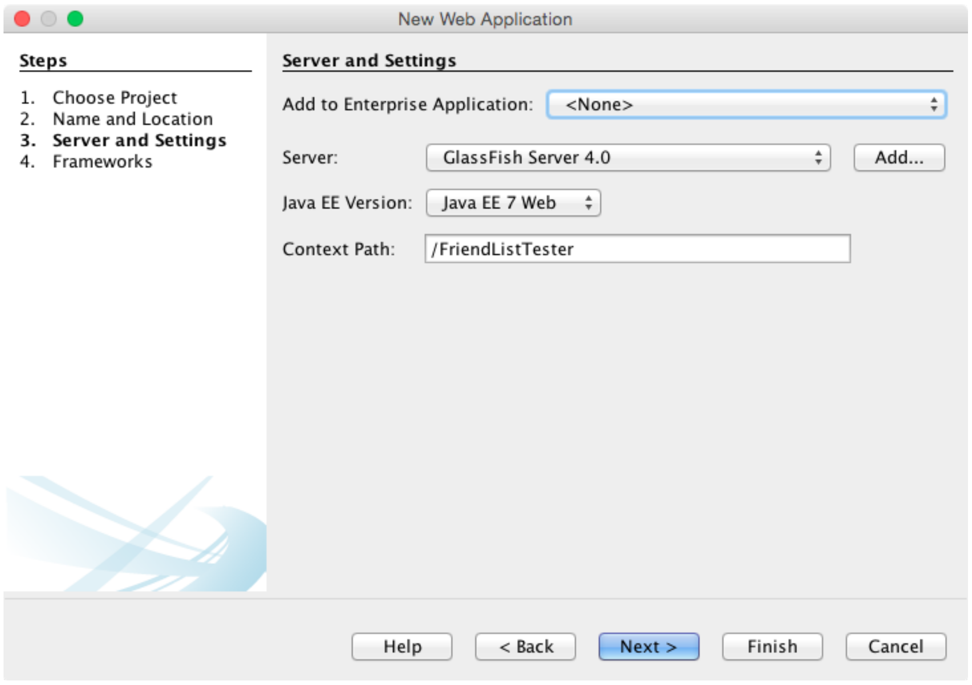
Project Folder: /NetBeansProjects/FriendListTester

Use Dedicated Folder for Storing Libraries

Libraries Folder:

Different users and projects can share the same compilation libraries (see Help for details).

Figure 44: Creating a new war project


New Web Application

Steps
1. Choose Project
2. Name and Location
3. **Server and Settings**
4. Frameworks

Server and Settings

Add to Enterprise Application: <None>

Server: GlassFish Server 4.0

Java EE Version: Java EE 7 Web

Context Path: /FriendListTester

Figure 45: Creating a new war project, step 2

6. Right-click on the war project, where you have created the RESTful Web Service and select “Test RESTful Web Services”.

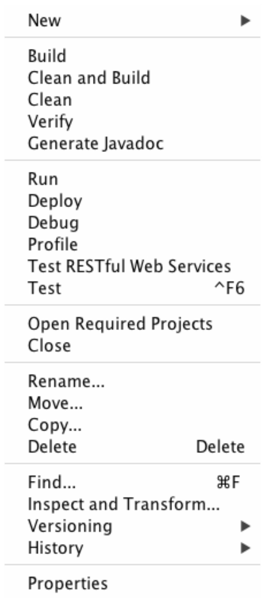
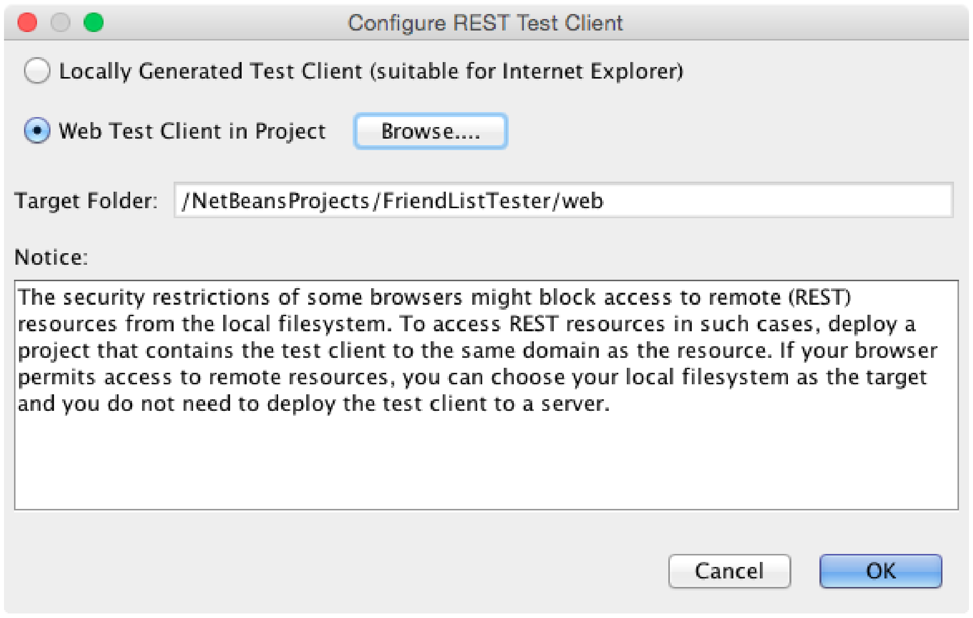


Figure 46: Test RESTful Web Services

Now, you have the option to generate the test client inside the service project or in another Java Web project. This option lets you work around security restrictions in some browsers. You can use any Web project, as long as it is configured to deploy in the same server domain as the service project. Select the `FriendListTester` project.


Configure REST Test Client

( ) Locally Generated Test Client (suitable for Internet Explorer)

(•) Web Test Client in Project Browse....

Target Folder: /NetBeansProjects/FriendListTester/web

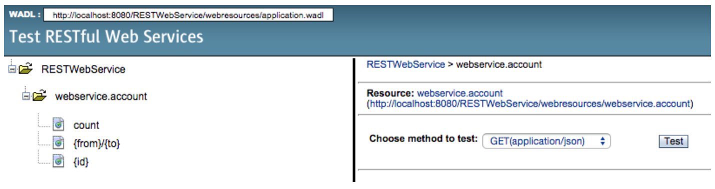
Notice:

The security restrictions of some browsers might block access to remote (REST) resources from the local filesystem. To access REST resources in such cases, deploy a project that contains the test client to the same domain as the resource. If your browser permits access to remote resources, you can choose your local filesystem as the target and you do not need to deploy the test client to a server.

Cancel OK

Figure 47: Configuration of the REST Test Client

7. Run both project and surf to `http://localhost:8080/FriendListTester/test-resbeans.html`. Figure 48 shows the generated web page. Click on `webservice.account` and then on the Test button in the right panel. This will query all `Accounts` in the database.


WADL : http://localhost:8080/RESTWebService/webresources/application.wadl

Test RESTful Web Services

RESTWebService
webservice.account
count
{from}/{to}
{id}

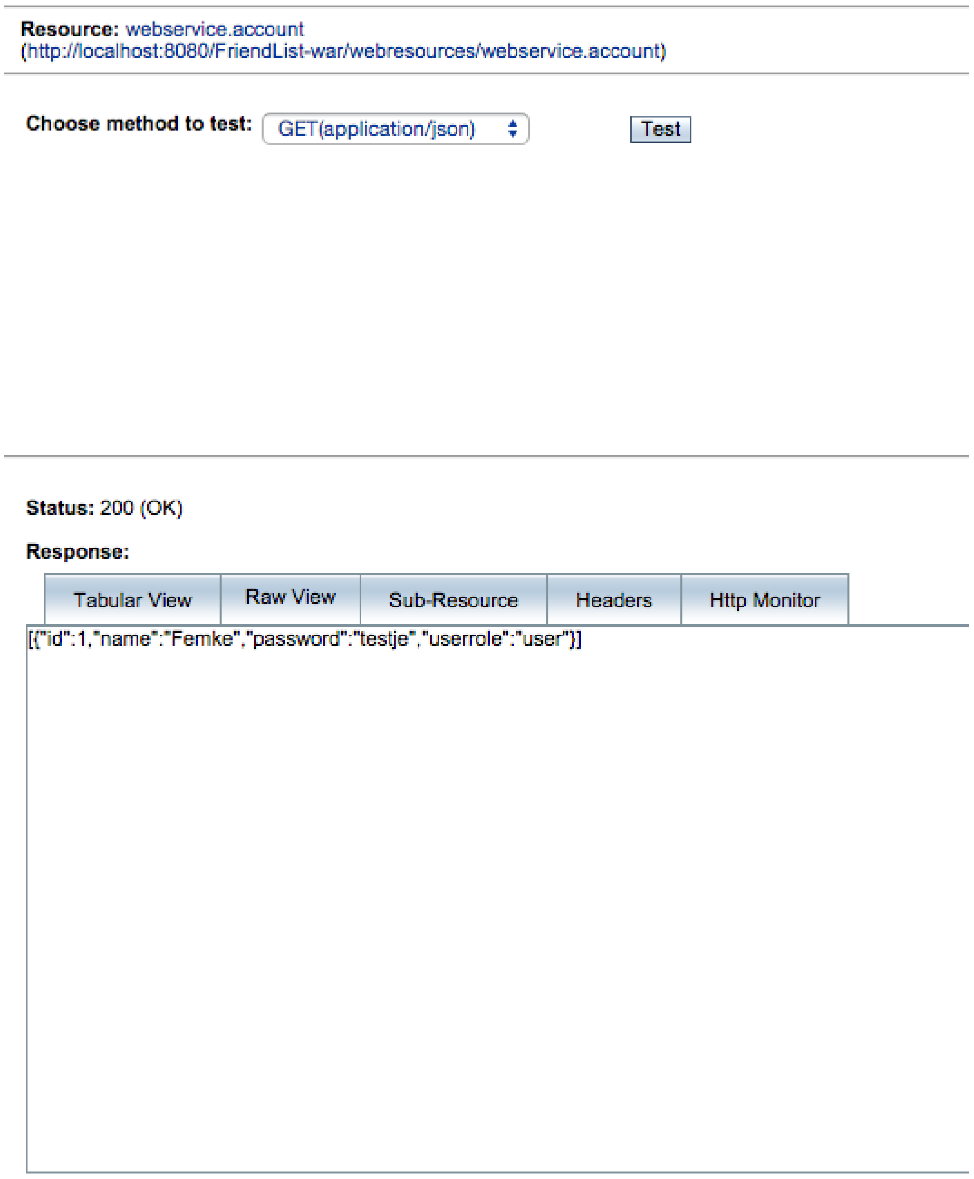
RESTWebService > webservice.account

Resource: webservice.account (http://localhost:8080/RESTWebService/webresources/webservice.account)

Choose method to test: GET(application/json) Test

Figure 48: Website to test the RESTful Web Service

Figure 49 shows the Status and the Response of the Web Service call.



Resource: webservice.account
(http://localhost:8080/FriendList-war/webresources/webservice.account)

Choose method to test: GET(application/json)

Status: 200 (OK)

Response:

```
[{"id":1,"name":"Femke","password":"testje","userrole":"user"}]
```

Figure 49: Result of the GET operation

There are 5 tabs in the Test Output section.

- The Tabular View is a flattened view that displays all the URIs in the resulting document. Currently this view only displays a warning that Container-Containee relationships are not allowed.
- The Raw View displays the actual data returned. Depending on which mime type you selected (application/xml or application/json), the data displayed will be in either XML or JSON format, respectively.
- The Sub Resource tab shows the URLs of the root resource and sub resources. When the RESTful web service is based on database entity classes, the root resource represents the database table, and the sub resources represent the columns.
- The Headers tab displays the HTTP header information.
- The HTTP Monitor tab displays the actual HTTP requests and responses sent and received.

8. Now, try to add a new `Account` to the database. Select the `post(applicationjson`, enter the JSON string, as shown below, and press Test. Status 204 is shown, as no content can be displayed.

```
{"id":"310","name":"Emiel","password":"test1214","userrole":"user"}
```

3.5 Calling a RESTful Web Service within a class

A RESTful Web Service can be called from within a Java class. The following code example executes the GET operation (make sure that you import `javax.ws.rs.client.*` and `javax.ws.rs.core.*`):

```
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target("http://localhost:8080/FriendList-war").path("
            webresources/" + id);
        Invocation.Builder invocationBuilder = target.request(MediaType.APPLICATION_JSON_TYPE);
        invocationBuilder.header("some-header", "true");
        Response response = invocationBuilder.get();
```

4 JSON

JSON[2] (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

On Minerva, you can find a jar file, containing a library, that can be used for processing JSON strings. More information on the classes and the API can be found on `http://www.json.org/java/`.

The `Response` generated in Section 3.5 can be transformed to JSON using the following code:

```
JSONObject json = new JSONObject(response.readEntity(String.class));
```

[2] http://www.json.org/