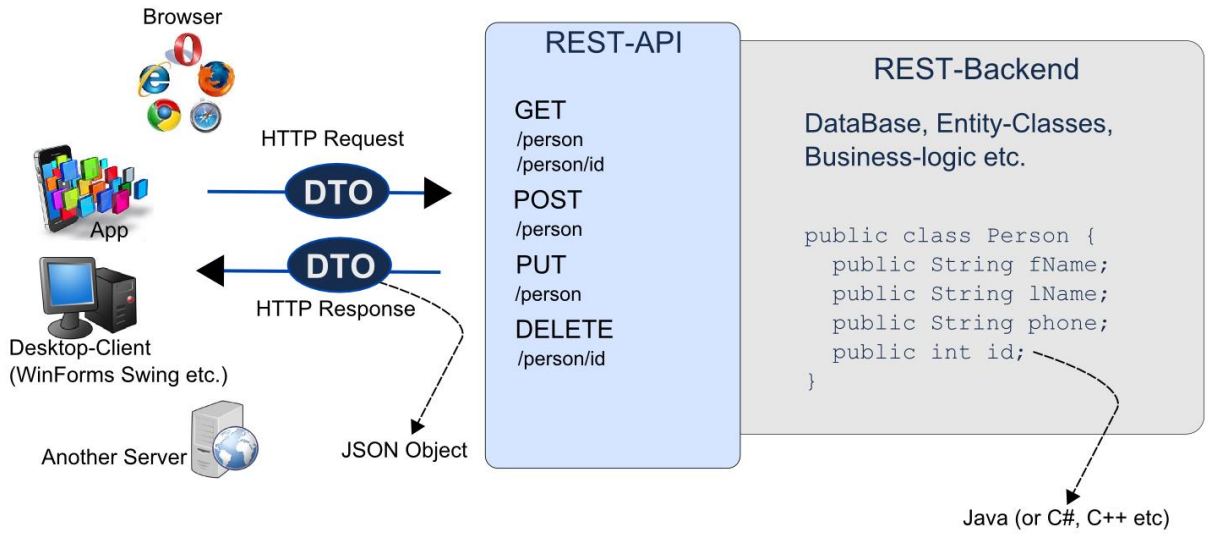
2) REST with JAX RS and Client with JQuery and AJAX

In this exercise we will go through most of the steps necessary to create a REST driven application as sketched below, using a very simple one-class model, to simplify matters.



In the backend we will implement a Java Person class and a façade implementing the following interface:

```
public interface IPersonFacade {
  Public void addEntityManagerFactory(EntytyManagerFactory emf)
  public Person addPerson(Person p);
  public Person deletePerson(int id);
  public Person getPerson(int id);
  public List<Person> getPersons();
  public Person editPerson(Person p);
}
```

By now you should know that an important REST constraint is to have a layered system, with Resources Decoupled from their Representation. For this exercise we will expose data as JSON using the URIs given in the figure above.

For the REST-URIs that either return or consumes a Person, the following JSON must be used:

```
{"fName":"Lars","lName":"Mortensen","phone":"12345678","id":0}
```

For the REST URI that creates a Person, use the JSON above, without the id property:

For the GET method that returns all Persons, the JSON must have this format:

```
[{"fName":"Lars","lName":"Mortensen","phone":"12345678","id":0},
{"fName":"Peter","lName":"Olsen","phone":"12345678","id":1}]
```

To help with the conversion between your Java backend and the JSON-based frontend you should design a utility class as sketched below:

```
public class JSONConverter {
  public static Person getPersonFromJson(String js){..}
  public static String getJSONFromPerson(Person p) {..}
  public static String getJSONFromPerson(List<Person> persons) {..}
}
```

Tasks

Server side:

1) Create a new NetBeans Maven Web Project
2) Create an Entity class (with a corresponding database) to implement the Person from the figure above
3) Create a script to setup some sample data and "call" the script from your persistence.xml file (HINT)
4) Implement a Façade class from IPersonFacade and use JUnit to test the Façade. Se hint-3 for help.
5) Implement and test the JSONConverter class introduced above
6) Implement the GET methods from the REST-API and test via a browser
7) Implement the POST method and test using Postman
8) Implement the PUT method and test using Postman
9) Implement the DELETE method and test using Postman

Client side:

10) Implement a read-only page to show all Persons in a table. The table must be built in the browser using JQuery and data fetched via a REST call.
11) Add a refresh button that should refresh the page designed in the previous step. Use Postman to add a new Person to verify that we actually get an updated list, without having to create a new page on the server.
12) Add an option to create new Persons (inspired by the figure below) on the same page as the one with the table. Use the REST API to create the new person on the server. Also don't rebuild the whole table when a new person is created, but add a new row to the table with the new person. Se *hint-4* for help

ID	First Name	Last name	Phone	
1	Peter	Olsen	1234	delete / edit
2	Hanne	Olsen	1234	delete / edit
3	Kurt	Wonnegut	7985476	delete / edit
4	Heidi	Petersen	1234	delete / edit

Reload Data | Add New Person

13) Add an option to delete a Person (row) as sketched on this figure (*see Hint-5*)
14) Add an option to edit a Person (row) as sketched on the figure.

Hints

Hint-1

Create the required JSON from a map entry, without a matching Java Class. (Requires the gson-xxx.jar file)

```
JsonObject quote = new JsonObject();
int key = 1; //Get the second quote
quote.addProperty("quote", quotes.get(key));
String jsonResponse = new Gson().toJson(quote);
```

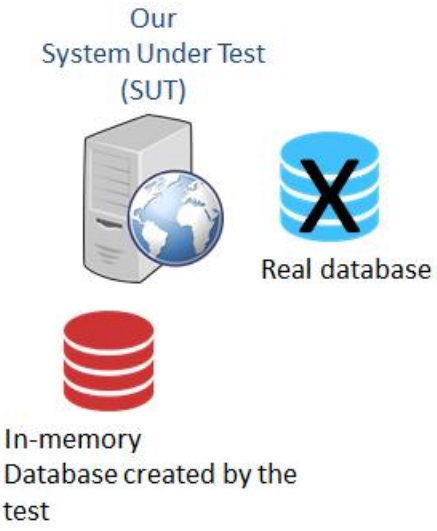
Hint-2

Get the quote from provided JSON

```
//Get the quote text from the provided Json
JsonObject newQuote = new JsonParser().parse(json).getAsJsonObject();
String quote = newQuote.get("quote").getAsString();
quotes.put(nextId++, quote);
```

Hint-3 Testing a JPA Design (mocking the database)

Mocking database operations the traditional way can be very cumbersome and time consuming.



If we limit our goals to: "*we want Database Unit Test Cases that can be executed immediately after a project is cloned, without the need to set up a local/external-test database*", we can "cheat" and use an in-memory database for testing

We can use the **Apache Derby Database** for this purpose, since it can be run embedded (in-memory), which means: No need for database setup, start, and external database files etc.
Setup via an entry like this in you POM:

```
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derby</artifactId>
  <version>10.12.1.1</version>
</dependency>
```

You setup this database via the persistence.xml file, just as usual. We can have more than one persistence-unit in your persistence.xml, as long as you supply each with a unique subpackage name. See below for how to setup and additional persistence-unit, that uses the derby in-memory database:

```
<persistence ver.....................................>
  <persistence-unit name="pu_development" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>entity.Person</class>
    <properties>
      ……..
      <property name="eclipselink.canonicalmodel.subpackage" value="development"/>
    </properties>
  </persistence-unit>
  <persistence-unit name="pu_test" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>entity.Person</class>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:derby:target/testDB;create=true"/>
      <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
      <property name="eclipselink.ddl-generation.output-mode" value="database"/>
      <property name="eclipselink.canonicalmodel.subpackage" value="test"/>
    </properties>
  </persistence-unit>
</persistence>
```

Hint-4: *For this task, you could create a* `Form` *next to the table (visible only when you press a "add Person" button or you could use Bootstrap to bring up a modal with the* `Form` (http://www.w3schools.com/bootstrap/bootstrap_modal.asp)

Hint-5: *One way to do this would be to use HTML-5s data attributes*[1]*, When you build the table and the anchor tag for the delete action, you could add the id to an attribute designed by you, with the id for the person in that row as value. The anchor tag for the first row in the table above, would look something like this*:

```
<a href="#" class="btndelete" data-personid="1">delete</a>
```

*With the class "**btndelete**" added to the anchor, we can attach an event handler to all these anchors and get the data-**personid** value as sketched in this exa3333mple:*

```
$(".btndelete").click(function (evt) {
  var id = $(this).data("personid");
  … });
```

[1] http://www.w3schools.com/tags/att_global_data.asp