Poul Klausen

JAVA 11

Web applications and Java EE

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 11: WEB APPLICATIONS AND JAVA EE

SOFTWARE DEVELOPMENT

Java 11: Web applications and Java EE: Software Development
1st edition

ISBN 978-87-403-1948-4
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

This book is the eleventh in a series of books on software development. The programming language is Java, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process and how to develop good and robust applications. This book deals with the development of web applications where the focus is on the server side and how to develop dynamic web pages. The book starts with an introduction to servlets, followed by a brief presentation of Java Server Pages. The rest of the book deals with how to write web applications using Java Server Faces, and after reading the book, you should be able to write classic web applications. However, the book contains little about the client side, which is dealt with first in the next book. The book is not a reference to JSF elements, and the aim is, through examples, to show what you are doing and it is necessary that you examine the online documentation to get an overview of the many JSF elements that exist.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in Java. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples and exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code to the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with one or two larger sample programs, which focus primarily is on process and an explanation of how the program is written. On the other hand appears the code only to a limited extent – if at all – and the reader should instead study the finished program code perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

The programming language is, as mentioned Java, and besides the books use the following products:

1. NetBeans as IDE for application development
2. MySQL to the extent there is a need for a database server (from the book Java 6 onwards)
3. GlassFish as a web server and application server (from the book Java 11 onwards)

It is products that are free of charge and free to install, and there is even talk about products, where the installation is progressing all by itself and without major efforts and challenges. In addition, there are on the web detailed installation instructions for all the three products. The products are available on Windows and Linux, and it therefore plays no special role if you use Linux or Windows.

All sample programs are developed and tested on machines running Linux. In fact, it plays no major role, as both Java and other products work in exactly the same way whether the platform is one or the other. Some places will be in the books where you could see that the platform is Linux, and this applies primarily commands that concerning the file system. Otherwise it has no meaning to the reader that the programs are developed on a Linux machine, and they can immediately also run under Windows unless a program refers to the file system where it may be necessary to change the name of a file.

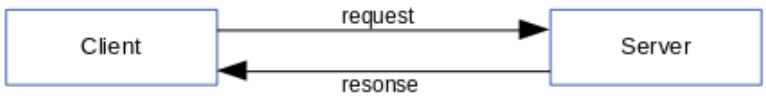
Finally a little about what the books are not. It is not “how to write” or for that matter reference manuals in Java, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

Everything in the previous books that has been said about system development, programming and Java has been aimed at applications running on a single computer – also called desktop applications. In the following books, I will treat the development of applications that run on multiple machines and exchange data over a network, which is often the Internet, and in fact, such programs are the most common in practice – at least if you look at programs used in companies or public institutions. Besides running the programs on a network, they are characterized by having many concurrent users and typically included in an IT solution that includes several programs that work together to solve the desired tasks. Development of such programs requires new technical solutions and, on the other hand, they make new demands for the system development process itself, but fortunately everything so far been said is still valid, but it is necessary to expand with new concepts.

In the literature, such programs are generally referred to as enterprise applications, and without it being precisely defined what it is, it covers programs for large companies and organizations, but perhaps you should think about it in that way that a program is not just a program but a solution to a work situation or task in a larger company. I will start by looking at the development of web applications, which are applications that run through the Internet, and although it's not the only kind of enterprise applications, and even maybe it's not even sharp anymore what are web application and what are not, then at least there is a talk of some technology and applications that are developed and run in a way other than traditional office programs, and in addition, they are programs that all meet, whether it's at work or privately.

Web applications have been around for many years and have, as a result of a long development, been more and more widely distributed from simple static websites to actual applications such as web stores and applications such as Google's office programs, but while there's been much happening, the principle is still the same with a server and a client:



The client is often an ordinary workstation with a browser. A browser is a program similar to any other PC program, but using the browser, the user can enter a *request* to a web server by entering a web address in the browser's address bar. A web server is also a program that runs on a machine somewhere. It is a program that is constantly running and listening for a request. A *request* means that the client's browser wants a HTML document to be sent back and the web server will then load the document and send it back to the client's browser as a *response*. Once the client gets the document, the browser will rendering it and display it on the screen. This means that the browser interprets the HTML code and based on the code determines how the document should appear on the screen.

It is at least the basic principle, and as it works in the beginning with *World Wide Web*, but today there is a lot more. A request regarding not only to send an HTML document back, but often data is sent along with the request (that could be data about items to be added to a shopping cart) and it may also be information to the server to find specific data (for example from search criteria to goods). This means that the server has to do a lot of other things like, for example, to save data in databases, retrieve data from a database and dynamically build the response to be sent to the client. In other words, on the basis of the client's requests, the web server must perform software, programs that are basically developed and executed like any other program, but only programs that run on the machine hosting the web server.

A web application is thus characterized by, that the program's code being executed on the server, which then sends the result back to the client in the form of a dynamically generated HTML document. However, a *client-side* program can also contains code that is executed by the browser. There are several options, but the most important is *JavaScript*, which is script code (which is just text) that is sent as part of the HTML document. Of course, it requires that the browser is able to interpret and execute the *JavaScript* code, but all modern browsers can. Another option is to send actual translated code to the browser, for example, a *Java applet* that is written and translated in exactly the same way as for example a Swing program. One might think of a *Java applet* as a Java application that runs in the browser window. Of course, it requires the browser to support the execution of a Java applet, and it usually requires installing a plugin (extension) for the browser. However, since a Java applet is translated into binary code and downloaded and executed on the client's machine, this technology entails a security risk that many are not happy about. Thus, a web application can include both code executed on the server side and code executed on the client side, but such that the largest part of the code is server code. One are therefore called web applications for *client-server* applications.

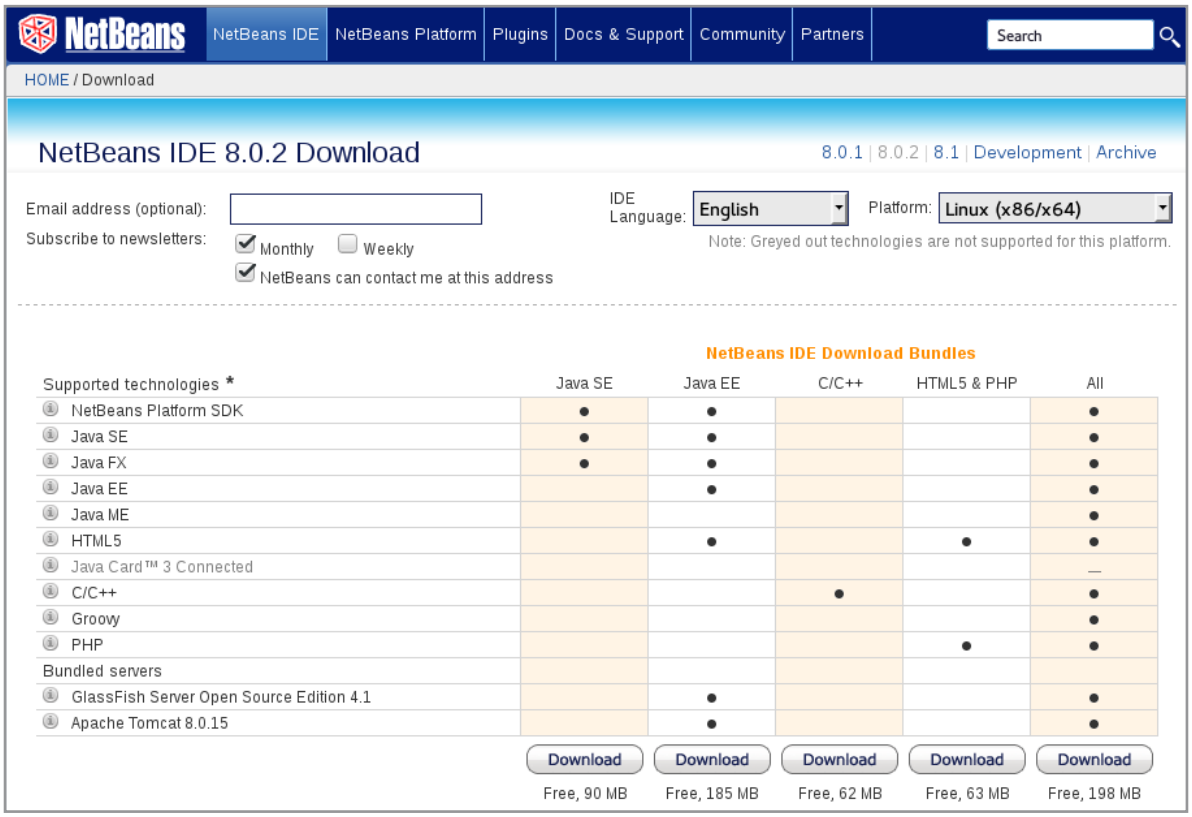
As described above, a web application builds on network traffic, and every time a user makes a request, there is a delay before getting a response. Web applications therefore run different from a traditional desktop application. Nevertheless, many of the tasks that previously were solved by programs running on the specific client machine are taken ower by web applications. There are several reasons for this. The most important thing is the obvious advantage that you can access data regardless of where you are located, and it is not necessary to install the program on the user's machine. Then there is the delay that draws in the other direction, but it does not mean the same as before and, among other things because that today we have a much higher rate on the Internet than before, but also because we have used to the fact that web applications works that way. Finally, technically, a number of measures have been taken to ensure that the delay is not felt so much primarily by ensuring that no more data is sent than necessary. One could say that it has always been the goal of getting a web application to appear to the user, as it was a usual program installed on the user's computer.

The goal of this book is to show how to develop web applications in Java. However, it requires a modest knowledge of HTML that I do not want to touch on, but although it should not be in place, you will probably be able to follow the development of the examples anyway, as it is relatively small that I use, and also the development tools (NetBeans) provides great help.

Finally, it should be mentioned that web applications are not the solution to everything and there are still programs that should be developed as classical PC applications, primarily because they can not live with the above delay. Now, a common PC application can also communicate with a server over the Internet and thus be part of an enterprise application, but it requires another technology that I return to in subsequent books. It should also be mentioned that handheld computers such as mobile phones today play a very important role and again set new requirements for the programmer, a topic that is dealt with in the the book Java 15.

1.1 THE DEVELOPMENT TOOL

In order to develop and test web applications, you must have a web server. There are many options, but in this case I need a web server that supports web applications developed in Java. So far all programs have been written using *NetBeans*, and when you download the product you can choose from several packages:

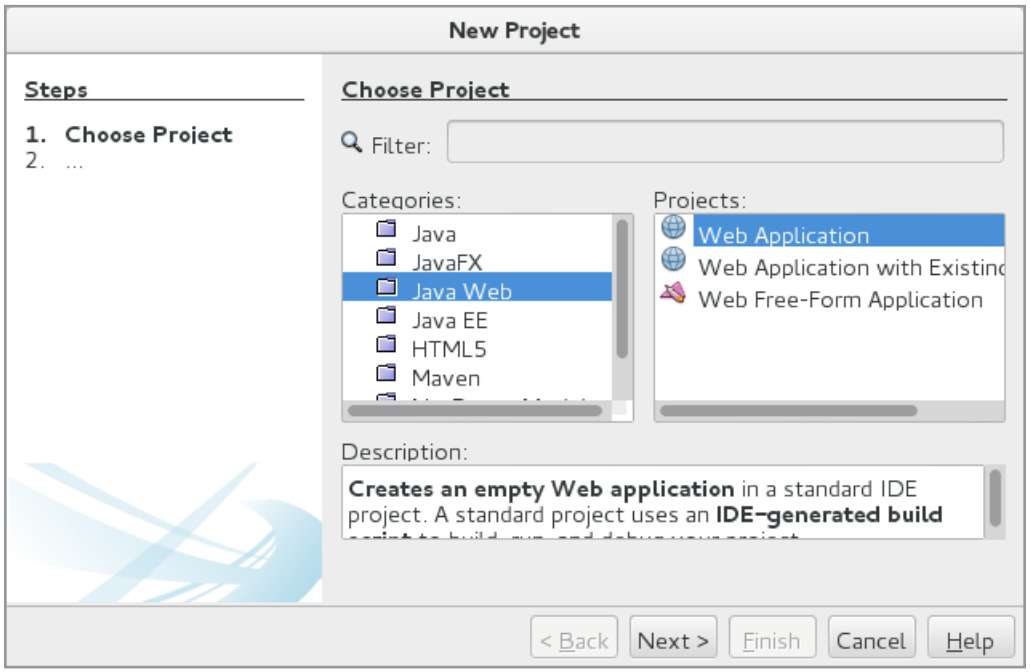


So far, I've used the first one, but now I need *Java EE*, which stands for *Java Enterprise Edition*. It expands with Java EE, which contains a lot of development tools for enterprise applications including web applications and *HTML5* that support the development of web applications that uses *HTML5*. Finally, two web servers are included. The difference is that the latter is a common web server that supports Java web applications and is sufficient for the sake of this book, but it is not an application server. That is the *GlassFish* server, and as it is necessary for the following books, I will use it everywhere. Both servers are included, but you can decide if you want to install both (there is no need to install *Tomcat* unless you want to experiment with it).

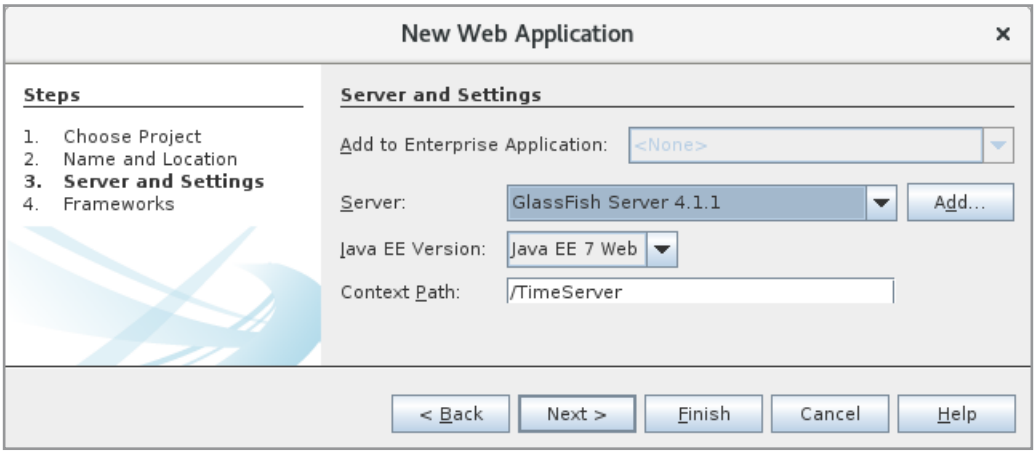
When you download *NetBeans* bundle with *Java EE*, you get a self-extracting script, and generally it is easy to install *Java EE* and *GlassFish*, but the book has an appendix that shows how.

2 SERVLET

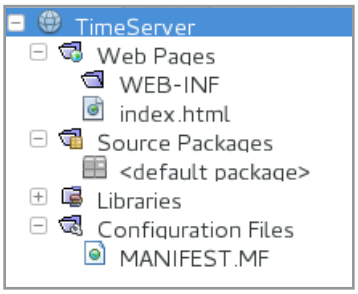
The basic concept of a Java web application is a servlet, which is a Java class that has methods that can generate response to the client in the form of HTML. Since it is a regular Java class, whose methods are performed on the server, it can perform anything a class can and it can, for example, read data in a database. A servlet can therefore dynamically generate a HTML document whose content depend on the situation. In fact, it is rare, in practice, you look at these servlets as you usually develop dynamic websites like *JSP* pages or *Facelets* (explained later), but in both cases the pages are translated to servlets, so this section about what a servlet is, but also how you write a servlet using *NetBeans*. After I opens NetBeans, I choose File | New Project:



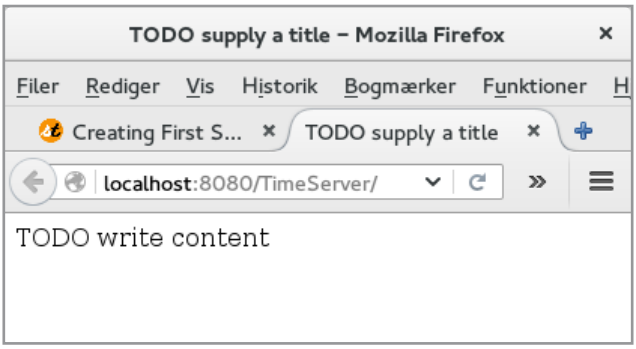
but this time, *Categories* must be *Java Web*, while *Projects* should be *Web Application*. When you click *Next*, you get the usual window where you need to enter a project name and select the folder where the project is to be created. I have called the project *TimeServer*. When you click *Next*, you get a window to select the server where the application is to be hosted:



It is probably already available for the *GlassFish* server, so there is no need to do anything and just click *Finish*. After that, a project for a web application (see below) has been created with folders for the different types of files that the project may consist of, but for the time being there is only one single file named *index.html*, which is a common HTML document, and it can be perceived as the application's home page:



I will add a few changes in a while, but it is a full-featured HTML document, and trying to run the *NetBeans* project opens the browser and displays the following page:

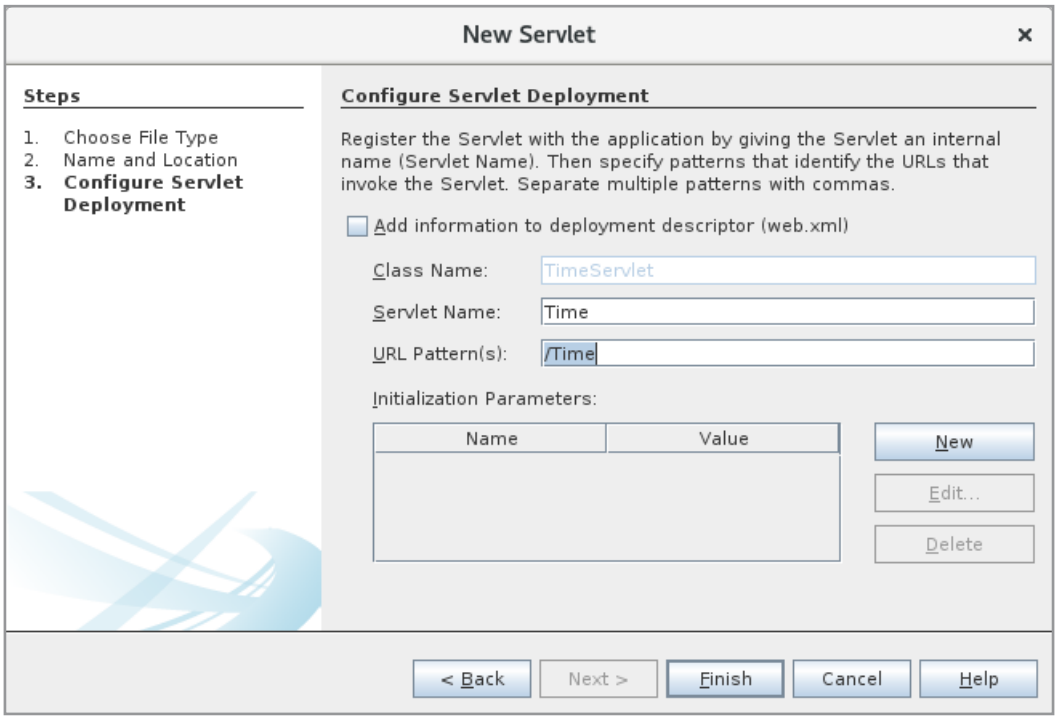


With regard to the browser window, note that the title bar displays the text from the document's *title* element, while the content of the page itself is the text from the body part of the document. Finally, note the address bar:

```
localhost:8080/TimeServer
```

localhost is the name of the local machine and hence the physical server where the web server is installed. As mentioned, a web server is a program that is running constantly, and such a program is out of the world known by the name of the server (which is actually a web address) and a *port number*. A web server can be configured to use a specific port number and *GlassFish* uses as default port 8080. In this case, the *TimeServer* application is called, and as no other is specified, the server will use the application's home page (send the home page as a response), which is *index.html*.

Currently, the application has no servlets, and I will now add a servlet. This happens by right clicking on <*Default package*> and selecting *New* | *Servlet*. In the following window I have called the servlet for *TimeServlet* (the window shows a warning that you can ignore in this place) and when you click *Next* you get the window, as shown below:



Here you should note that I have changed both *Servlet Name* and *URL Pattern (s)*. It is not necessary and is most done to show that it is allowed. Then click on *Finsh*, and NetBeans creates a servlet (where I have deleted some of the comments):

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name = "Time", urlPatterns = {"/Time"})
public class TimeServlet extends HttpServlet
{
 /**
  * Processes requests for both HTTP <code>GET</code> and <code>POST</code>
  * methods.
  * @param request servlet request
  * @param response servlet response
  * @throws ServletException if a servlet-specific error occurs
  * @throws IOException if an I/O error occurs
  */
 protected void processRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
 {
  response.setContentType("text/html;charset=UTF-8");
  try (PrintWriter out = response.getWriter())
  {
   /* TODO output your page here. You may use following sample code. */
   out.println("<!DOCTYPE html>");
   out.println("<html>");
   out.println("<head>");
   out.println("<title>Servlet TimeServlet</title>");
   out.println("</head>");
   out.println("<body>");
   out.println(
    "<h1>Servlet TimeServlet at " + request.getContextPath() + "</h1>");
   out.println("</body>");
   out.println("</html>");
  }
 }
```

```
 /**
  * Handles the HTTP <code>GET</code> method.
  * @param request servlet request
  * @param response servlet response
  * @throws ServletException if a servlet-specific error occurs
  * @throws IOException if an I/O error occurs
  */
 @Override
 protected void doGet(HttpServletRequest request,
HttpServletResponse response)
  throws ServletException, IOException
 {
  processRequest(request, response);
 }

 /**
  * Handles the HTTP <code>POST</code> method.
  * @param request servlet request
  * @param response servlet response
  * @throws ServletException if a servlet-specific error occurs
  * @throws IOException if an I/O error occurs
  */
 @Override
 protected void doPost(HttpServletRequest
request, HttpServletResponse response)
  throws ServletException, IOException
 {
  processRequest(request, response);
 }

 /**
  * Returns a short description of the servlet.
  * @return a String containing servlet description
  */
 @Override
 public String getServletInfo()
 {
  return "Short description";
 }
}
```

The comments explain to some extent what the individual methods do, but I will explain a little more in a while. However, you must note that in the class definition there is an annotation:

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

which tells me that it's a servlet, what its name and URL are, and here you should especially note that these are the names that I have chosen when the servlet in question was created. Initially, I will return to *index.html* (the home page) that I have changed to the following:

```
<!DOCTYPE html>
<html>
 <head>
  <title>TODO supply a title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
 </head>
 <body>
  <h1>A simple html document</h1>
  <h2><a href="Time">What time is it?</a></h2>
 </body>
</html>
```

You should notice the *h2* element that enclose a link that refers to my servlet. If you now run the application, it opens a browser:

It's not that strange, but clicking on the link you will get another window:

You must remark the address bar, which tells you that it is the result of the servlet that appears.

I will now look a little closer to the code of the servlet class. First, note that the class inherits *HttpServlet*, which is the base class of a servlet. If you consider the method *processRequest()*, it is clear that it creates a HTML document. The method has two parameters, which represents respectively a *request* (from a browser) and a *response*. The communication between client and server occurs after the *HTTP* protocol, which is a simple text-based protocol. When the browser executes a request, it can be done in two ways, as are called *GET* and *POST*, and the main difference is how the browser sends data together with the request. When the browser sends a request to a servlet, one of the methods *doGet()* and *doPost()* are performed, and both methods have two parameters, which are references to objects that represent respectively a request and a response. In this case, nothing else happens than these objects are sent to the method *processRequest()*.

It initializes the response object to an HTML document, after which it determines a reference *out* to the object's writer. The reference is used to print text for the document, text that in this case constitutes a HTML document. When the *try* block goes out of scope, the reference is deleted, which means that the document is terminated and a response is send to the client.

Now I will change the method *processRequest()* as follows:

```
protected void processRequest(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException
{
 response.setContentType("text/html;charset=UTF-8");
 try (PrintWriter out = response.getWriter())
 {
  out.println("<!DOCTYPE html>");
  out.println("<html>");
  out.println("<head>");
  out.println("<title>Servlet TimeServlet</title>");
  out.println("</head>");
  out.println("<body>");
  out.println("<h1>The time is: " + getTime() + "</h1>");
  out.println("</body>");
  out.println("</html>");
 }
}

private String getTime()
{
 Calendar date = Calendar.getInstance();
 return String.format("%d. %s %d %02d:%02d:%02d", date.get(Calendar.DATE),
  getMonth(date.get(Calendar.MONTH)), date.get(Calendar.YEAR),
  date.get(Calendar.HOUR), date.get(Calendar.MINUTE), date.get(Calendar.SECOND));
}

private String getMonth(int n)
{
 switch (n)
 {
  case 0: return "January";
  case 1: return "February";
  case 2: return "March";
  case 3: return "April";
  case 4: return "May";
  case 5: return "June";
  case 6: return "July";
  case 7: return "August";
  case 8: return "September";
```

```
  case 9: return "October";
  case 10: return "November";
  case 11: return "December";
 }
 return "";
}
```

In fact, only one statement has been changed, but it now calls the method *getTime()*, which returns a string with the current date and time. The goal of all is to show that a servlet can contain all the Java code that may be needed, as a servlet is simply a Java class. Another goal is that my servlet is now dynamic so that the html sent to the browser depends on when the application is being executed. Below is an example after clicking the link on the home page:

Note that you do not have to open the page by clicking the link on the home page, but you can also directly open the page – and get the servlet that creates the page – by entering the address directly in the browser.

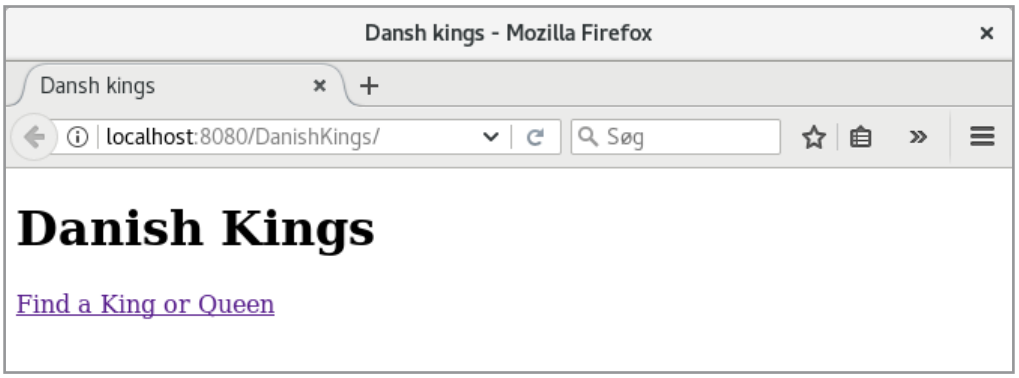
Once you have written a web application as above, it must be hosted on the application server before it can be opened in the browser. The task is called *deployment*, and it was earlier and can still be complex, but if the application is developed using NetBeans, it's all about itself as part of the build process without the need to do anything. If you examine the project, you will see that under the dist folder is a file with (in this case) the name *TimeServer.war* and it is a package that contains the files that the deployment process requires. I will later return to the deployment process as more complex applications may require more actions, and especially if the application is to be hosted on a server on another machine. If you opens the browser and enter the address field

```
localhost:8080/TimeServer
```

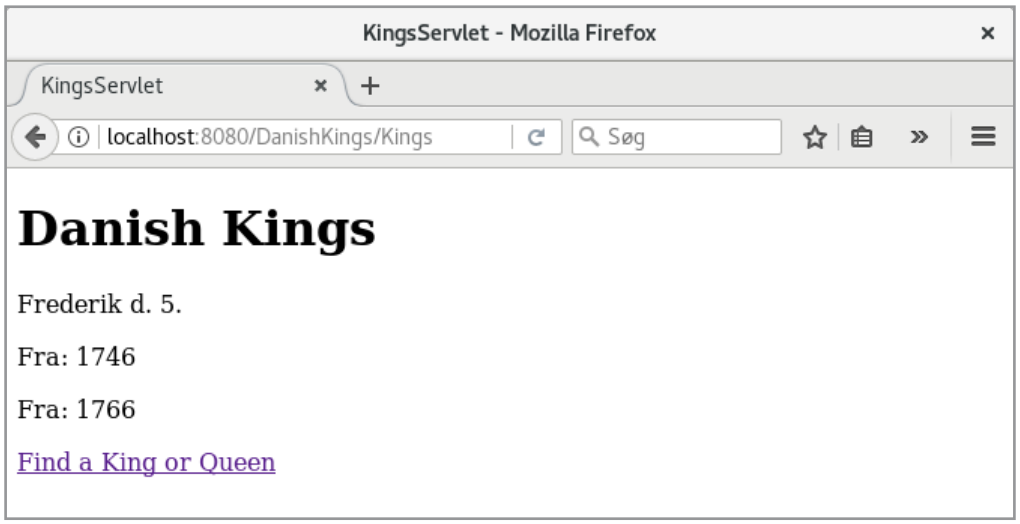
the application's home page opens. This corresponds to sending a request to the local *GlassFish* server asking for it to perform the *TimerServer* application, and *GlassFish* has then sent as response the application's start pages, which are *index.html*.

EXERCISE 1

In the same way as above, write a web application that uses a servlet. The application must start with a page (have a home page) as shown below:



If you click on the link, the result could be:

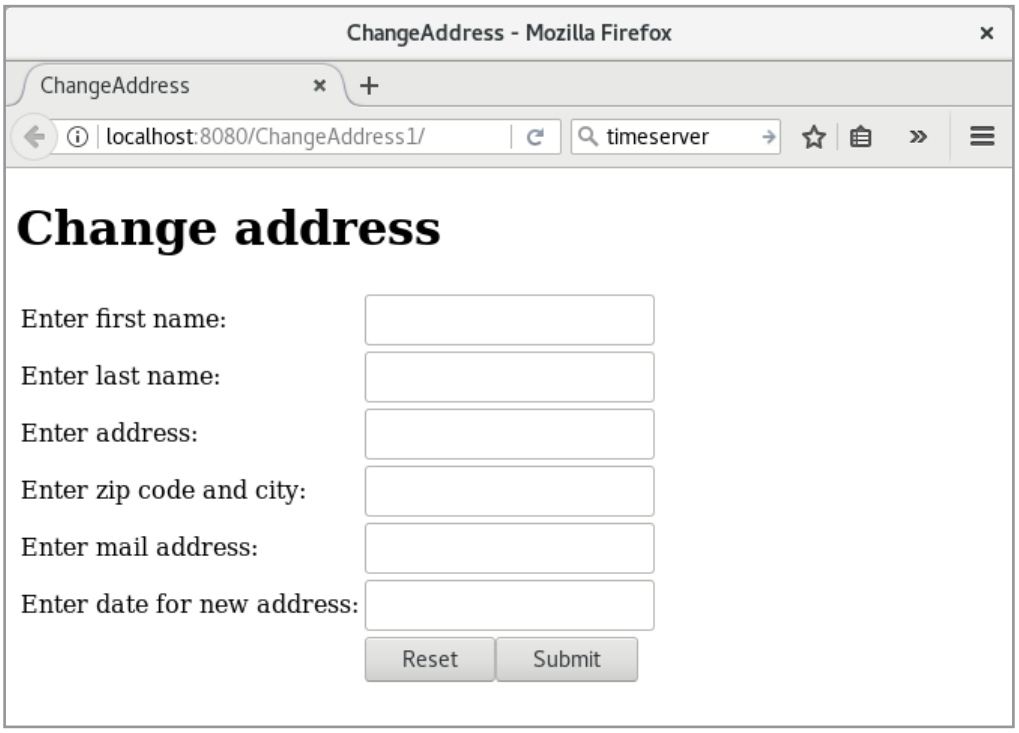


which shows the name and reign of a random Danish king. If you click on the link again, you will have a name and a reign of another random king.

Start by creating a new *Web Application* project and adjust the home page *index.html* to show the first page. Then add a servlet whose response is the other page. This servlet needs a list with the kings. You can find it in the file *kings*, which is a text file, and you can put the content of the file as a static array into your servlet class.

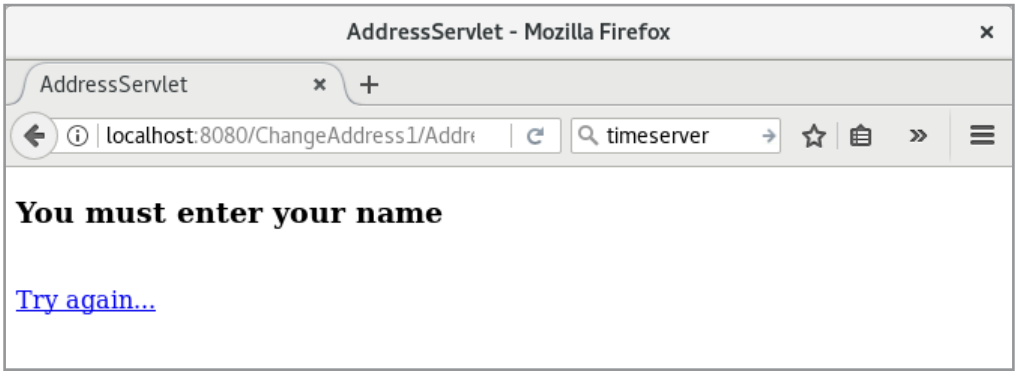
2.1 CHANGE ADDRESS 1

As another example of a servlet, I will show a web application where the user must fill out a form that can be regarding as a moving message. The entered information is then sent to the server where they are validated and if they can be accepted, a receipt will appear that the message that has been received. Otherwise, the user will be prompted to re-enter the message. If you open the application in the browser, you get the following window:


ChangeAddress - Mozilla Firefox
ChangeAddress
localhost:8080/ChangeAddress1/
timeserver
Change address
Enter first name:
Enter last name:
Enter address:
Enter zip code and city:
Enter mail address:
Enter date for new address:
Reset
Submit


It's only an opportunity if you act on it
IKEA.SE/STUDENT
IKEA
everyone deserves good design
© Inter IKEA Systems B.V. 2009

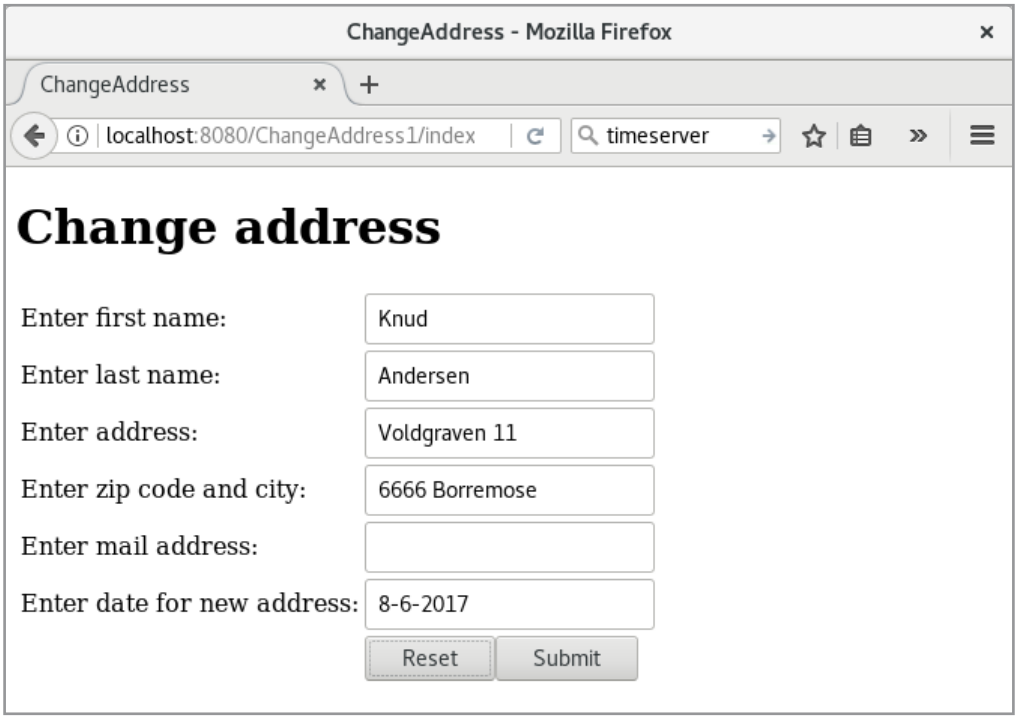
If you click on the Submit button, the form data is sent to the server, which could, for example, answer back with the following message:

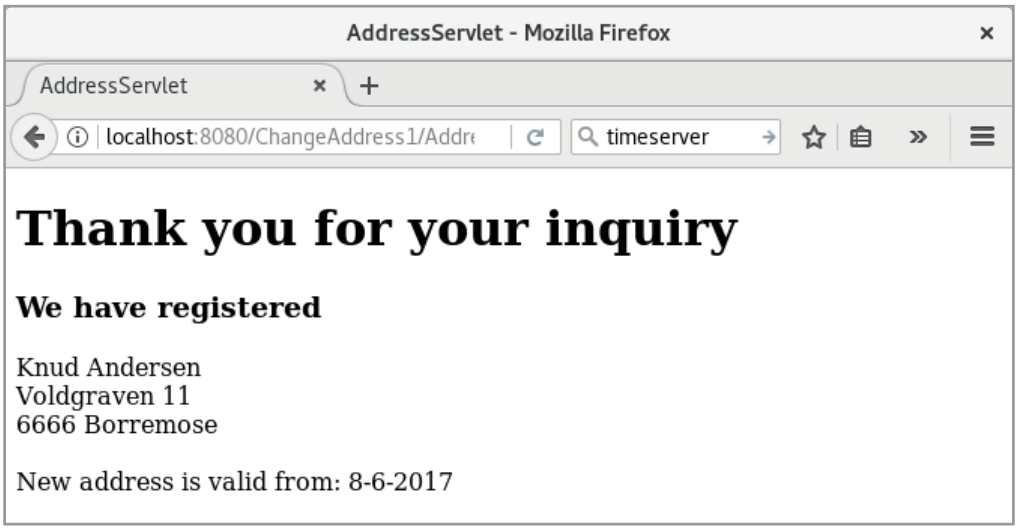


since the user has not entered the entire address. You will then have to click on the link, and you will be able to fill out the form again.

If you fill in all fields as shown below, where the field for the email address is not filled out, and where the date is entered on the form DD-MM-YYYY (that is, hyphen between the fields) and then click *Submit*, you will receive a receipt as a response (see below).

It is, of course, a very simple application, yet it is an application that sends data from the client to the server where they are processed, after which the client gets a response and in many ways it is the principle of many web applications.





The basis is a NetBean project, called *ChangeAddress1*, which as the first project in this chapter is a Web Application project. NetBeans creates a home page, called *index.html*, which I have modified to the following:

```
<!DOCTYPE html>
<html>
 <head>
  <title>ChangeAddress</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
 </head>
 <body>
  <h1>Change address</h1>
  <form method="post" action="AddressServlet">
   <table>
    <tr>
     <td><label>Enter first name: </label></td>
     <td><input type="text" id="fname" name="fname"/></td>
    </tr>
    <tr>
     <td><label>Enter last name: </label></td>
     <td><input type="text" id="lname" name="lname"/></td>
    </tr>
    <tr>
     <td><label>Enter address: </label></td>
     <td><input type="text" id="addr" name="addr"/></td>
    </tr>
    <tr>
     <td><label>Enter zip code and city: </label></td>
     <td><input type="text" id="post" name="post"/></td>
    </tr>
```

```
    <tr>
      <td><label>Enter mail address: </label></td>
      <td><input type="text" id="mail" name="mail"/></td>
    </tr>
    <tr>
      <td><label>Enter date for new address: </label></td>
      <td><input type="text" id="date" name="date"/></td>
    </tr>
    <tr>
      <td></td>
      <td><input type="reset" value="Reset"/>
        <input type="submit" value="Submit"/></td>
    </tr>
   </table>
  </form>
 </body>
</html>
```

It is this page that shows the form. A *form* is defined in HTML with a *form* element. In this case, the form's fields are placed in a table, and it is alone to have the fields placed nicely underneath each other in two columns. The table has 7 rows, and the first 6 rows have a label and an *input* element. A label shows a text, whereas an *input* element that in this case has the type of *text* is an input field. Each input field has a name and thus an identifier defined by an *id* and a *name* attribute, respectively. In this case, only the last one is necessary, but it is slightly different when an element is identified by an *id* or a *name*, but when the elements values are sent to the server as part of a form, it is the *name* attribute that is used. Should you, however, refer to an element from a style sheet, it is the *id* attribute that is used. Therefore, it may be a good idea to indicate both. The last row in the table also has two input components, but they have different types. Both components are rendered by the browser as a button, and the *value* attribute specifies the text that the button shows. The first component clear the form's input fields while the other (*submit* component) submits the form to the server.

If you consider the *form* element, it has two attributes. *action* indicates to which page on the server the request should be send to. In this case, it is *AddressServlet*, which is a servlet. The other method indicates what kind of request it is and here it is *POST*. This means that the elements of the form are sent together with the request as key/value pair, where the key is the value of the *name* attribute while value is the value of the *value* attribute, which for an input field is the entered text.

As mentioned, *AddressServlet* refers to a servlet. It is created in the same way as in the previous example, and I have called it *AddressServlet*, and on the *Configure Servlet Deployment* screen, I have not changed the values for *Servlet Name* and *URL Pattern (s)* this time, but I have created a package for the servlet. The completed code is shown below, where I have deleted all comments as well as the last method *getServletInfo()*:

```
package changeaddress.servlets;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import java.util.*;
import java.text.*;
import java.util.regex.*;
```

```
@WebServlet(name = "AddressServlet", urlPatterns = {"/AddressServlet"})
public class AddressServlet extends HttpServlet
{
 protected void processRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
 {
  response.setContentType("text/html;charset=UTF-8");
  request.setCharacterEncoding("UTF-8");
  try (PrintWriter out = response.getWriter())
  {
   out.println("<!DOCTYPE html>");
   out.println("<html>");
   out.println("<head>");
   out.println("<title>AddressServlet</title>");
   out.println("</head>");
   out.println("<body>");
   String fname = request.getParameter("fname");
   String ename = request.getParameter("lname");
   String addr = request.getParameter("addr");
   String post = request.getParameter("post");
   String mail = request.getParameter("mail");
   String date = request.getParameter("date");
   if (fname.length() == 0 || ename.length() == 0)
   {
    out.println("<h3>You must enter your name</h3><br>");
    out.println("<a href=\"index.html\">Try again…</a>");
   }
   else if (addr.length() == 0 || post.length() == 0)
   {
    out.println("<h3>You must enter the address</h3><br>");
    out.println("<a href=\"index.html\">Try again…</a>");
   }
   else if (mail.length() > 0 && !isMail(mail))
   {
    out.println("<h3>You must enter a legal mail address</h3><br>");
    out.println("<a href=\"index.html\">Try again…</a>");
   }
   else
   {
    try
    {
     SimpleDateFormat formatter = new SimpleDateFormat("dd-mm-yyyy");
     Date dato = formatter.parse(date);
     out.println("<h1>Thank you for your inquiry</h1>");
     out.println("<h3>We have registered</h3>");
     out.println(fname + " " + ename + "<br>");
     out.println(addr + "<br>");
```

```
      out.println(post + "<br>");
      out.println(mail + "<br>");
      out.println("New address is valid from: " + date);
     }
     catch (Exception ex)
     {
      out.println("<h3>You must enter a date on the form DD-MM-YYYY</h3><br>");
      out.println("<a href=\"index.html\">Try again…</a>");
     }
    }
    out.println("</body>");
    out.println("</html>");
  }
}

private boolean isMail(String mail)
{
 return Pattern.compile(".....").matcher(mail).matches();
}
```

```
 @Override
 protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
 {
  processRequest(request, response);
 }

 @Override
 protected void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
 {
  processRequest(request, response);
 }
}
```

In principle, it does not contain anything new to the first example, and almost the method *processRequest()* is also generated by NetBeans. Note that I have added a method *isMail()*, which is a method from earlier to validate an email address using a regular expression (I have not shown the code). The method *processRequest()* works in principle in the same way as in the first example, and dynamically creates the HTML to be sent as response. First, note the statement

```
request.setCharacterEncoding("UTF-8");
```

It is necessary for Danish letters to be decoded correctly. Next, you should notice how to grasp the form's values, for example:

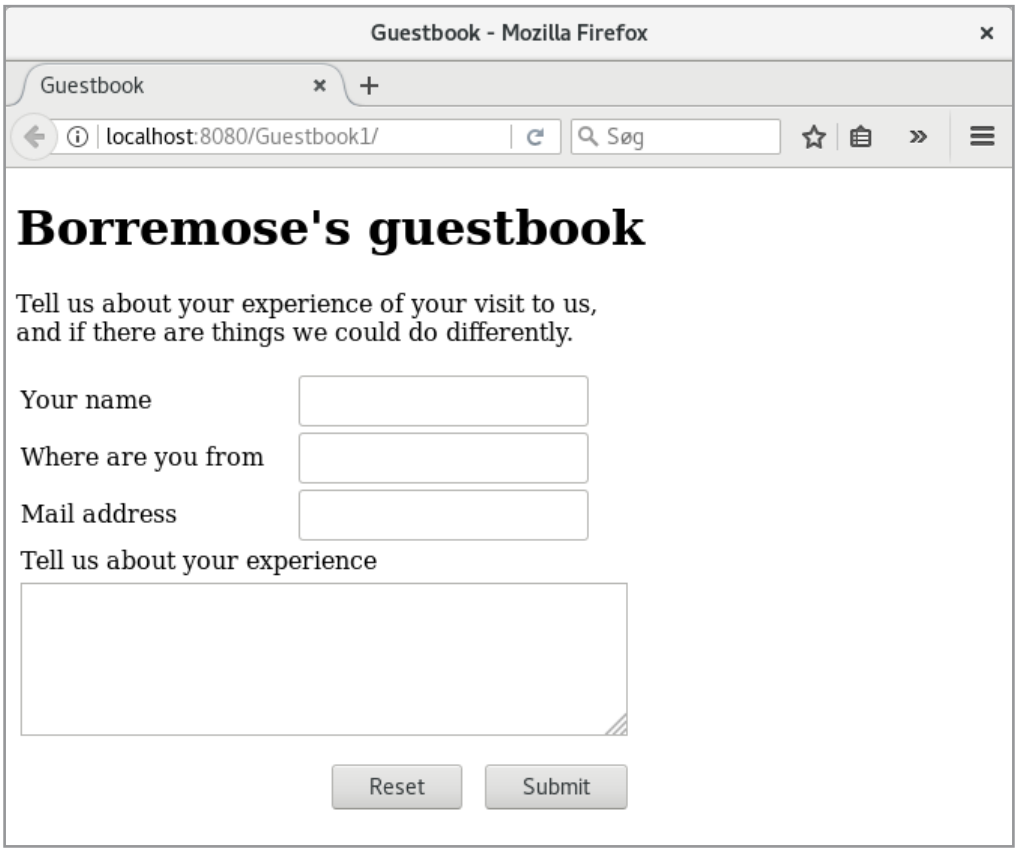
```
String fname = request.getParameter("fname");
```

that determines the value entered as the first name. The rest is, in principle, just off the road and consists in validating the fields sent from the form and, in the case of errors, sending an error message as a response. Thus, the server response is dynamically because it depends on the data sent.

It is clear that the servlet could do so much else and it could, for example, save the submitted data in a database. It would alone be a question to add some Java code.

EXERCISE 2

You must write a web application similar to the *ChangeAddress1* example. The application should this time show a page with a form for an electronic guestbook. When the application starts, it must display the following page:



where a guest can enter his name and where to come from (for example a city name). In addition, you can enter an email address as well as a text that tells about the visit. Name and where to come from must be entered, but you do not have to enter the email address, but if you do, it should be a legal address. The text may also be omitted.

Once you have completed the form and clicked on the *Submit* button, the form must be submitted to a servlet. If the form is not filled in correctly, you must receive an error message:

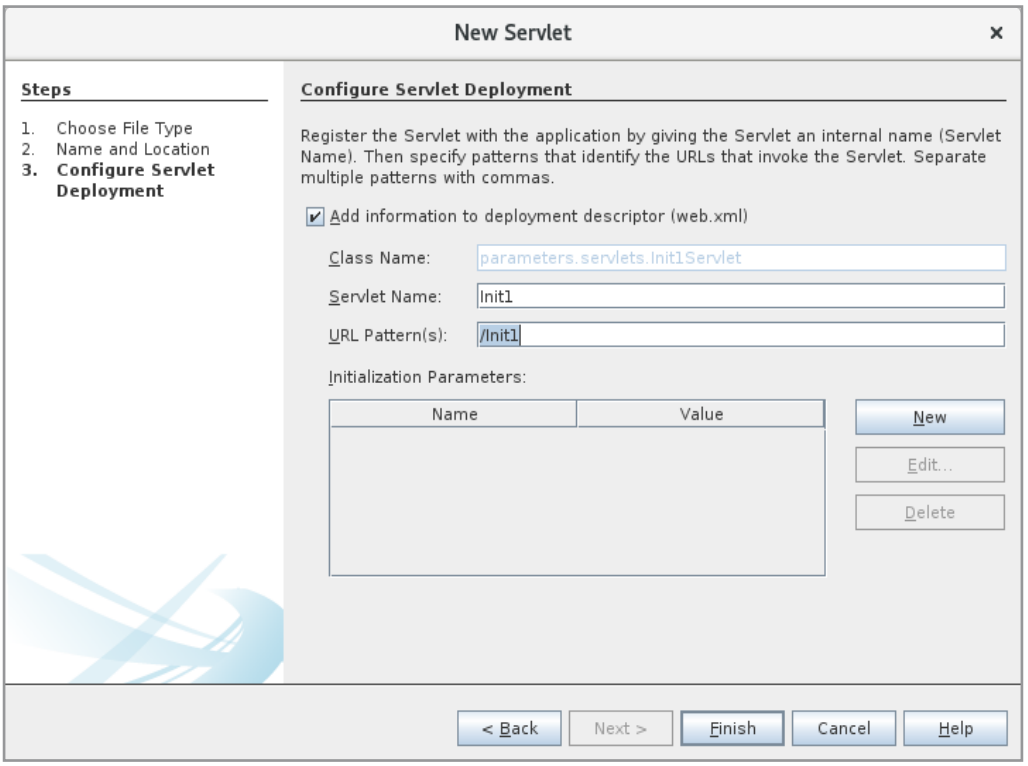
and you must be able to reopen the form so you can re-enter. If the form is correctly filled out, the user must receive the following receipt:

3 PARAMETERS AND SESSIONS AND MORE

Looking at the above examples, it is clear that one can reach far with regard to the development of dynamic websites and web applications alone by using html documents and servlets, but it is also clear that for a complex site it can be extensive to write a servlet and make it all look real. It's something that JSP and JSF have to solve, but before I show what it is, I'll show you some examples of servlets. Not so much for servlets, but more to put some basic concepts regarding web applications in place.

3.1 PARAMETERS TO SERVLETS

I will start by creating a new web application that I have called *Parameters*, and the result is again an application with a homepage called *index.html*. I will first show how to transfer parameters to a servlet when initialized. I have added a servlet named *Init1Servlet*, and in the configuration window I have defined the following (note the checkmark for *Add information to deployment descriptor (web.xml)*):



After the corresponding servlet has been created, NetBeans has created an XML document under the *WEB-INF* folder:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
 http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
 <servlet>
  <servlet-name>Init1</servlet-name>
  <servlet-class>parameters.servlets.Init1Servlet</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>Init1</servlet-name>
  <url-pattern>/Init1</url-pattern>
 </servlet-mapping>
 <session-config>
  <session-timeout>
   30
  </session-timeout>
 </session-config>
</web-app>
```

It is a configuration file that is used in connection with deployment of the application. You must note that the name of my servlet is *Init1* as selected in the above window and that it is a name that refers to the class *parameters.servlets.Init1Servlet*. You should also note that the servlet in the application has the address (URL) */Init1*.

I have next expanded the XML document, as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
 <servlet>
  <servlet-name>Init1</servlet-name>
  <servlet-class>parameters.servlets.Init1Servlet</servlet-class>
  <init-param>
   <param-name>owner</param-name>
   <param-value>Torus data</param-value>
  </init-param>
 </servlet>
 <servlet-mapping>
  <servlet-name>Init1</servlet-name>
  <url-pattern>/Init1</url-pattern>
```

```
 </servlet-mapping>
 <session-config>
  <session-timeout>
   30
  </session-timeout>
 </session-config>
</web-app>
```

This means that I have defined a key/value pair with the name *owner* and the value *Torus data*. I have then changed the code for *Init1Servlet* to the following, where I have only shown the first part of the class:

```
public class Init1Servlet extends HttpServlet
{
 protected void processRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
 {
  response.setContentType("text/html;charset=UTF-8");
  try (PrintWriter out = response.getWriter())
  {
   out.println("<!DOCTYPE html>");
   out.println("<html>");
   out.println("<head>");
```

```
    out.println("<title>Servlet InitServlet1</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Greetings from " +
     getServletConfig().getInitParameter("owner") + " 1</h1>");
    out.println("</body>");
    out.println("</html>");
   }
 }
```

Here you should note that the class is not decorated by a *@WebServlet* attribute. The reason is that the servlet in question is this time defined in the configuration file. The use of annotations is an alternative to the configuration file that makes it easier to write a servlet (you do not need to maintain the xml configuration file), but conversely, changes require that the code may be translated again while the configuration file can be maintained without the need to modify the servlet code. You should also note that in *processRequest()* I have only changed a single line (the third last statement) and that it retrieves the parameter defined in the XML document.

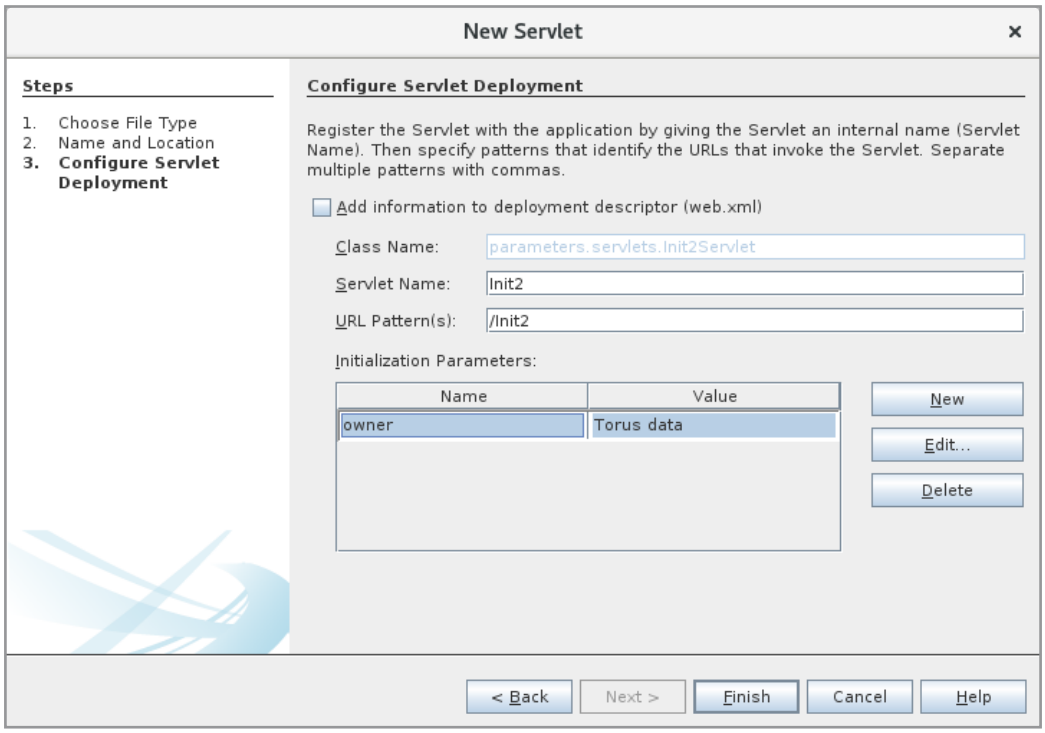
I have then changed the start page to the following:

```
<!DOCTYPE html>
<html>
 <head>
  <title>Parameters</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
 </head>
 <body>
  <h1>Parameters to servlets</h1>
  <p><a href="Init1">Init1</a></p>
 </body>
</html>
```

and executing the application and clicking on the link, you get the window

Here you should note how to transfer parameters to a servlet using the xml configuration file.

If you look at NetBean's *Configure Servlet Deployment* window above, the bottom option is to specify initialization parameters for a servlet. This can be used instead of manually editing the XML document as shown above. I have added another servlet, this time called *Init2Servlet*, and I have configured deployment as follows:



The important thing is that there is no tick in *Add information to deployment descriptor (web.xml)* and an *Initialization Parameter* has been added. This means that web.xml is not updated, but if you look at the code, the class definition is decorated:

```
@WebServlet(name = "Init2", urlPatterns = {"/Init2"},
 initParams={@WebInitParam(name="owner", value="Torus data") })
public class Init2Servlet extends HttpServlet
{
```

It is used during deployment as an alternative to *web.xml* and you can see how the name and URL are defined. Finally, I have added a parameter. The code for *processRequest()* is essentially the same and after adding another link on the start page, you can make a request to the new servlet and the result is as follows:

3.2 SESSIONS

When you request a web page from the browser and the server answers back with a response, it will be done as already mentioned in accordance with the HTTP protocol. This protocol is basically stateless, which means that after the server has replied back with the response, the server has "forgotten" all about the request, and next time a request for the same page is received, the server will perceive it as a new request. This model fits badly with modern web applications, and you can, among other things, solve the problem using the *session* concept.

When a user opens the browser and sends a request to a new page, the server will investigate whether a *session ID* is included. It does not matter if it is a request for a new page entered in the browser's address bar, and the server will then generate a *session ID* and send it with the response back to the client. If the client then performs a submit or, for example, clicks on a link that refers to a page in the same web application, the browser will send the session ID with, and the server will include it in its response. That way, the term called a *session* has been introduced that exists as long as the server refers to pages within the same web application.

This can be used to define data that lives throughout the session. When the server creates a session ID, it also creates a session object and this object contains a collection for *Object* objects. The server can thus create and save objects in this collection, and these objects live as long as the session object lives, and therefore they can be used by all pages within that session. It is important to note that the objects are only known on the server side and are not sent to the client. The only one sent to the client is the session ID, which is sent back and forth for each request/response pair.

The individual session objects live on the server until they are explicitly removed or until they are removed due to a timeout (for example, it may be 30 minutes). This means, on the other hand, that the server can have many session objects that are no longer used, thus using space, and partly that session objects (such as a shopping cart) may disappear if you stop working.

To illustrate the use of sessions, I have created a web application called *Sessions*. For this application I will add three servlets. When adding a servlet to a web application, it should not be placed in the project's default package (what has been done above in the first example). Typically, I create a package named *projectname.servlets* (in this case, *sessions.servlets*) and places my servlets there. In this case, I have also created a package called *sessions.data*, and I have added the following class that can represent a person with a name and a job title:

```
package sessions.data;

public class Person
{
 private String name;
 private String title;

 public Person(String name, String title)
 {
  this.name = name;
  this.title = title;
 }
```

```
 public String getName()
 {
  return name;
 }

 public String getTitle()
 {
  return title;
 }
}
```

You should note that it is a fairly common model class. As a next step, I have added the following servlet:

```
@WebServlet(name = "CreateServlet", urlPatterns = {"/CreateServlet"})
public class CreateServlet extends HttpServlet
{
 protected void processRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
 {
  response.setContentType("text/html;charset=UTF-8");
  request.setCharacterEncoding("UTF-8");
  try (PrintWriter out = response.getWriter())
  {
   String name = request.getParameter("name");
   String title = request.getParameter("title");
   if (name != null && title != null)
   {
    sessions.data.Person pers =
     new sessions.data.Person(name.trim(), title.trim());
    HttpSession session = request.getSession(true);
    session.setAttribute("person", pers);
   }
   out.println("<!DOCTYPE html>");
   out.println("<html>");
   out.println("<head>");
   out.println("<title>Servlet SaveServlet</title>");
   out.println("</head>");
   out.println("<body>");
   out.println("<h3>Your session object is now updated</h3>");
   out.println("<h3>Session id: " + request.getSession().getId() + "</h3>");
   out.println("<a href=\"index.html\">Go to start</a>");
   out.println("</body>");
   out.println("</html>");
  }
 }
```

The method *processRequest()* starts by determining the values of two form fields from the page that sent a request to this servlet. Is it possible, and they are both different from *null*, they are used to create a *Person* object. Next, a reference to the session object is added:

```
HttpSession session = request.getSession(true);
```

This means that the object must be created if it does not already exist. As the next step, the *Person* object is added to the session object's collection with the key *person*. Finally, a response is returned, which includes the session object ID.

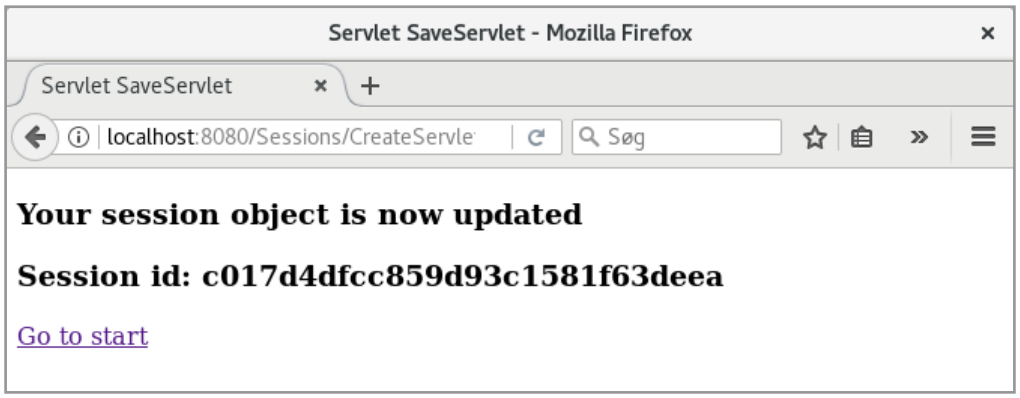
The code for the start page is the following:

```
<!DOCTYPE html>
<html>
 <head>
  <title>ServletTest</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
 </head>
 <body>
  <h3>Start page</h3>
```

```
  <form method="post" action="CreateServlet">
   <table>
    <tr>
     <td>Enter name</td>
     <td><input type="text" name="name" id="name" /></td>
    </tr>
    <tr>
     <td>Enter job title  </td>
     <td><input type="text" name="title" id="title" /></td>
    </tr>
    <tr>
     <td colspan="2"><input type="submit" value="OK" /></td>
    </tr>
   </table>
  </form>
  <ol>
   <li><a href="NameServlet">Show name</a></li>
   <li><a href="TitleServlet">Show title</a></li>
  </ol>
 </body>
</html>
```

If you run the application you get the following window:

If you enter a name and job title here and click OK, you get the following results:



It tells that the session object is updated, as well as what the session ID is. You should note that it is represented as a large hexadecimal number, which you generally can not use for anything, but in this case you can use it to see when a new session is started.

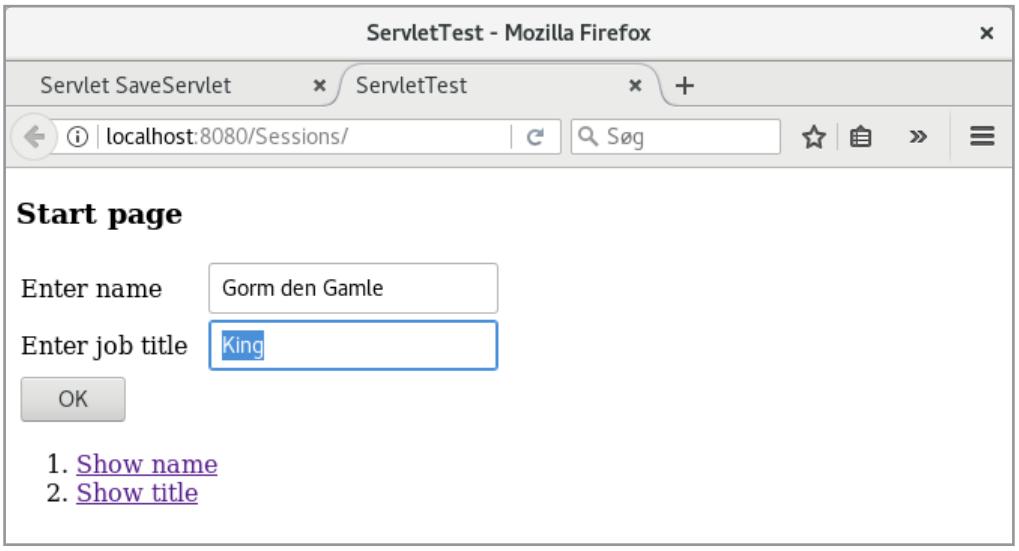
As a next step, a servlet has been added, which shows the name of the person that was created:

```
@WebServlet(name = "NameServlet", urlPatterns = {"/NameServlet"})
public class NameServlet extends HttpServlet
{
 protected void processRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
 {
  response.setContentType("text/html;charset=UTF-8");
  try (PrintWriter out = response.getWriter())
  {
   String name = null;
   try
   {
    name = ((sessions.data.Person)
      request.getSession().getAttribute("person")).getName();
   }
   catch (Exception ex)
   {
    name = ex.toString();
   }
   out.println("<!DOCTYPE html>");
   out.println("<html>");
   out.println("<head>");
   out.println("<title>NameServlet</title>");
   out.println("</head>");
   out.println("<body>");
```

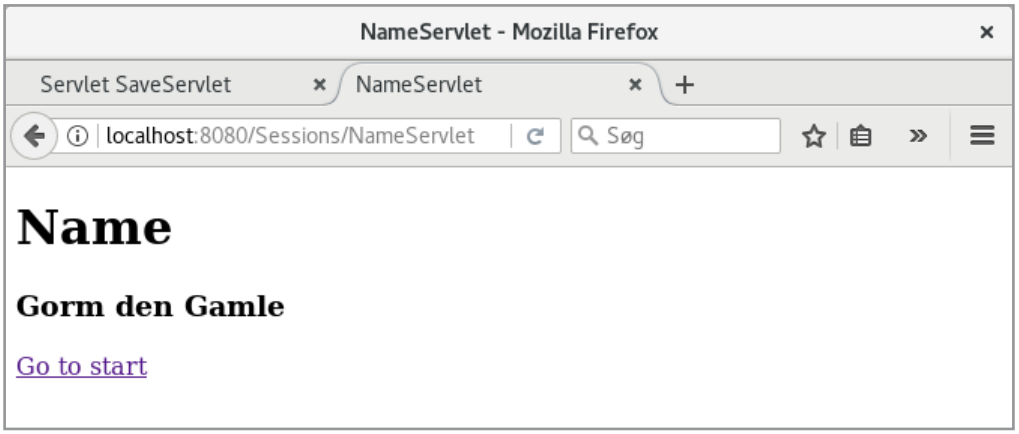
```
    out.println("<h1>Name</h1>");
    out.println("<h3>" + name + "</h3>");
    out.println("<a href=\"index.html\">Go to start</a>");
    out.println("</body>");
    out.println("</html>");
  }
}
```

There is not much to explain, but you should notice how to refer to the session object and to the object that is stored under the name *person*. The result is an *Object*, and therefore a type of cast is necessary.

Similarly, a servlet named *TitleServlet* has been created, which sends a response with the *Person* object's title. If you on the start page enter:



and clicking *OK* and then from the start page clicking on the *Show name* link, you get the window:



3.3 REDIRECTION

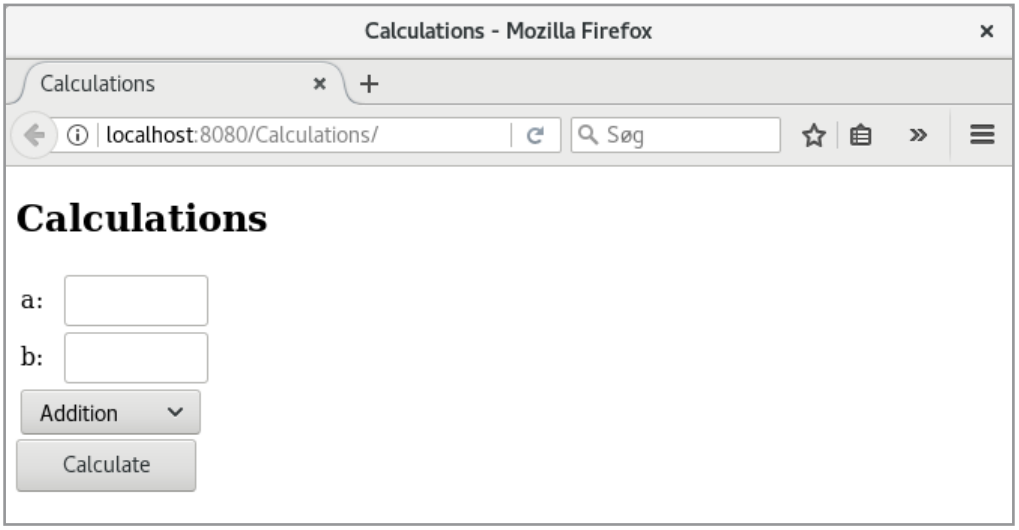
If you send a request to a servlet, it can perform a data processing and then send the request to another servlet (or another web page). This can be done either as *redirect* or *forward*. In the first case, the server insertes a code into the HTTP header along with the new URL (in the *Location* field) and sends a response to the browser. It will then perform a request for the new URL. In the second case, the server will simply send the same request to another URL, including all attributes of key/value pairs from *form* fields. Thus, a forward does not involve the browser, and the address bar in the browser does not change as everything happens on the server side. I will show you how to perform a redirect and a forward, respectively. The starting point is the following HTML document, called *index.html* and is the start page of the application *Calculations*:

```
<!DOCTYPE html>
<html>
 <head>
  <title>Calculations</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
 </head>
 <body>
  <h2>Calculations</h2>
  <form method="post" action="MathServlet">
   <table>
    <tr>
     <td>a:  </td>
     <td><input type="text" name="numbera"
      style="width: 80px; height:20px; padding:0"/></td>
    </tr>
    <tr>
     <td>b:  </td>
     <td><input type="text" name="numberb"
      style="width: 80px; height:20px; padding:0"/></td>
    </tr>
    <tr>
     <td colspan="2">
      <select name="operation" style="width: 120px; height:30px; padding:0">
       <option value="add">Addition</option>
       <option value="sub">Subtraction</option>
       <option value="mul">Multiplikation</option>
       <option value="div">Division</option>
      </select>
     </td>
    </tr>
   </table>
   <input type="submit" value="Calculate"
    style="width: 120px; height:35px; padding:0"/>
  </form>
 </body>
</html>
```

If you run the application you get a page as shown below. Here you can enter two numbers, and there is also a list where you can choose one of the four calculations. If you click on the button, there is a submit of the form to a servlet *MathServlet*. This servlet does nothing but deciding which operation to be performed, and then passes that request to one of four additional servlets (*AddServlet*, *SubServlet*, *MulServlet* and *DivServlet*), but in the first two cases it happens as *redirect*, while in the other two cases it happens as *forward*. The four servlets for the calculations send a response as a simple page that shows the result. In principle, they are all the same, and as an example, *processRequest()* for *AddServlet* is shown below:

```
protected void processRequest(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException
{
 response.setContentType("text/html;charset=UTF-8");
 try (PrintWriter out = response.getWriter())
 {
  out.println("<!DOCTYPE html>");
  out.println("<html>");
  out.println("<head>");
  out.println("<title>Servlet AddServlet</title>");
  out.println("</head>");
  out.println("<body>");
  try
  {
   String a = request.getParameter("numbera");
   String b = request.getParameter("numberb");
   double s = Double.parseDouble(a) + Double.parseDouble(b);
   out.println("<p>" + a + " + " + b + " = " + s + "</p>");
  }
  catch (Exception ex)
  {
   out.println("<p>Illegal arguments</p>");
  }
```

```
  out.println("<a href=\"index.html\">Back to start</a>");
  out.println("</body>");
  out.println("</html>");
 }
}
```



The most interesting is *MathServlet*:

```
protected void processRequest(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException
{
 String opr = request.getParameter("operation");
 if (opr.equals("add") || opr.equals("sub"))
 {
  String a = request.getParameter("numbera");
  String b = request.getParameter("numberb");
  String site = opr.equals("add") ? "AddServlet" : "SubServlet";
  site += "?numbera=" + a + "&numberb=" + b;
  response.sendRedirect(site);
 }
 else if (opr.equals("mul") || opr.equals("div"))
 {
  ServletContext context = getServletConfig().getServletContext();
  RequestDispatcher dispatcher =
   opr.equals("mul") ? context.getRequestDispatcher("/MulServlet") :
   context.getRequestDispatcher("/DivServlet");
  dispatcher.forward(request, response);
 }
 else response.sendRedirect("index.html");
}
```

It starts by determining the value of the list and hence the form field *select* called *operation*. If it is *add* or *sub*, a redirect must be made to either *AddServlet* or *SubServlet*. However, it causes a problem as the browser will redirect to that servlet as a GET, which means that the form fields are not sent. Therefore, I must add them manually as parameters to the URL, which occurs by initializing a variable, which could for example have the value

```
AddServlet?numbera=33&numberb=51
```

With this address in place, a redirect is performed using the method *sendRedirect()*, which corresponds to the above address being returned to the browser, which then performs an HTTP GET to this address.

If the operation is not *add* or *sub*, a *ServletContext* object is determined that is used to create a *RequestDispatcher* object for a *MulServlet* or *DivServlet*. This object has a *forward()* method, which is used to forward the request to that servlet. When this happens, the request is forwarded, including all forms of fields and without involving the browser.

You are encouraged to test the application and, in the same context, note what happens to the browser's address bar.

3.4 COOKIES

As the last thing about servlets, I will show you how to create and use a cookie. A cookie is text sent back and forth between server and browser and can be used in the same way as session variables to maintain state information. Below is a servlet that creates a cookie:

```
@WebServlet(name = "SetServlet", urlPatterns = {"/SetServlet"})
public class SetServlet extends HttpServlet
{
 private static java.util.Random rand = new java.util.Random();

 protected void processRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
 {
  response.setContentType("text/html;charset=UTF-8");
  try (PrintWriter out = response.getWriter())
  {
   Cookie cookie =
    new Cookie("randomvalue" + rand.nextInt(10),"" +
    rand.nextInt(Integer.MAX_VALUE));
   cookie.setHttpOnly(true);
   cookie.setMaxAge(30);
```

```
        response.addCookie(cookie);
        out.println("<!DOCTYPE html>");
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet SetServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Cookie created...</h1>");
        out.println("<a href=\"GetServlet\">Show cookie</a>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

A cookie is represented by a *Cookie* object. In this case is created a cookie with the name *randomValue* plus a digit (such as *randomValue5*) and is assigned a random value. Subsequently, it is defined as a temporary cookie known by the browser and finally defined as having a 30-second timeout. If the number is negative, it means that it is removed with the current session.

Below is another servlet that reads the cookie:

```
@WebServlet(name = "GetServlet", urlPatterns = {"/GetServlet"})
public class GetServlet extends HttpServlet
{
 protected void processRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
 {
  response.setContentType("text/html;charset=UTF-8");
  try (PrintWriter out = response.getWriter()) {
   out.println("<!DOCTYPE html>");
   out.println("<html>");
   out.println("<head>");
   out.println("<title>Servlet GetServlet</title>");
   out.println("</head>");
   out.println("<body>");
   out.println("<h1>Servlet GetServlet at " + request.getContextPath() +
    "</h1>");
   out.println("<table><tr><td>Name</td><td>Value</td></tr>");
   Cookie[] cookies = request.getCookies();
   for(Cookie c: cookies)
   {
    out.println("<tr><td>");
    out.println(c.getName());
    out.println("</td><td>");
    out.println(c.getValue());
    out.println("</td></tr>");
   }
   out.println("</table>");
   out.println("<p><a href=\"index.html\">Set cookie</a></p>");
   out.println("</body>");
   out.println("</html>");
  }
 }
```

Here's not much to explain, but you should notice how to loop over all cookies – for the current session. Finally, there is the start page:

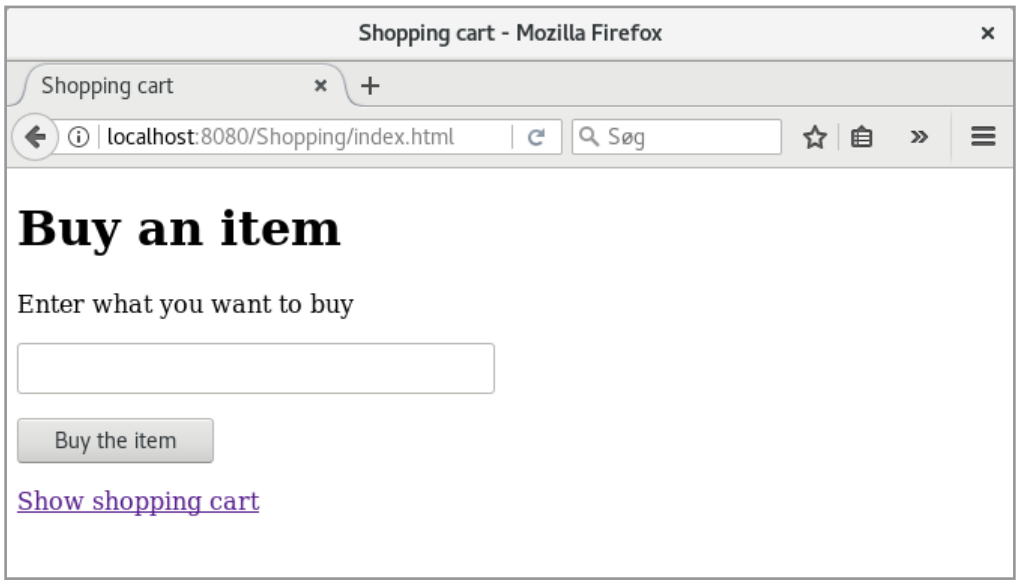
```
<!DOCTYPE html>
<html>
 <head>
  <title>Cookies</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
 </head>
 <body>
  <h1>Cookies</h1>
  <p><a href="SetServlet">Set cookie</a></p>
 </body>
</html>
```

If you run the program, you will find that more cookies are created, but also that a cookie changes value if you create a cookie with a name that already exists. You will also find that the individual cookies may disappear as there is a 30-second timeout.

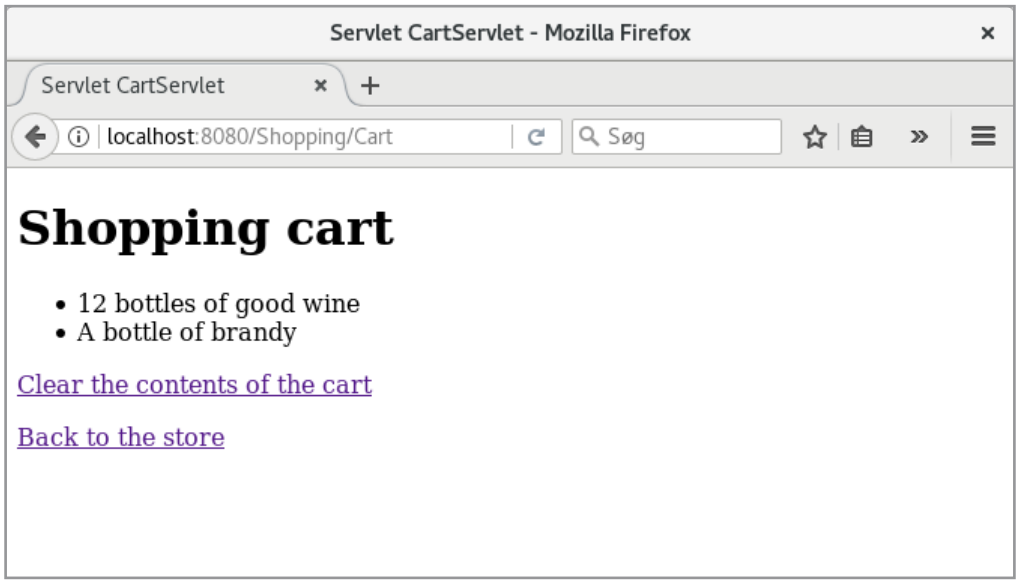
EXERCISE 3

You must write a web application that can simulate a very simple shopping cart. The purpose is primarily to show an example of the use of a session variable.

Start with a new Web Application project, which you can call *Shopping*. The home page *index.html* should be a simple form with a single entry field:



If you enter a product (a text) and press the button, the text must be sent to a servlet that saves it to an *ArrayList<String>*, that is a session item, after which the user is sent back to the home page. If you click on the bottom link, you must request a servlet that dynamically creates a page that shows the contents of the shopping cart – if there is anything in the basket. Otherwise, you just have to know that the basket is empty. The page with the shopping cart must also have a link that deletes the contents of the basket. You can solve this by referring the link to a servlet that deletes the contents of the session object. Below is an example of how a shopping cart of goods could look after 2 items have been added to the basket:


Servlet CartServlet - Mozilla Firefox
Servlet CartServlet
localhost:8080/Shopping/Cart
Søg
Shopping cart
12 bottles of good wine
A bottle of brandy
Clear the contents of the cart
Back to the store

4 JAVABEANS

In the following I will often refer to a *Java Bean*, and therefore little about what it is (in fact, I have referred to Java Beans in a previous book). In fact, it is nothing but a usual class, but it must fulfill a very few decisions:

1. A Java Bean must be *serializable*.
2. A Java Bean must have a default constructor.
3. All instance variables must be *private*.
4. Instance variables can have *get-* and *set* methods, and they must adhere to the names of the conventions so that if there is a *get* method *getName()*, the corresponding *set* method must be called *setName(…)*.

Apart from that, a Java Bean can be anything, and in the same way as other classes it may have various other methods and so forth. As an example, is below shown a Java Bean:

```
package dateserver;

import java.util.*;

public class DateBean implements java.io.Serializable
{
 private String date;
 private String time;

 public DateBean()
 {
  Calendar dt = Calendar.getInstance();
  date = String.format("%02d-%02d-%04d", dt.get(Calendar.DATE),
   dt.get(Calendar.MONTH) + 1, dt.get(Calendar.YEAR));
  time = String.format("%02d:%02d:%02d", dt.get(Calendar.HOUR_OF_DAY),
   dt.get(Calendar.MINUTE), dt.get(Calendar.SECOND));
 }

 public String getDate()
 {
  return date;
 }

 public String getTime()
 {
  return time;
 }
```

```
public void setDate(String date)
{
 try
 {
  StringTokenizer tk = new StringTokenizer(date, "-");
  if (tk.countTokens() != 3) throw new Exception("Illegal date");
  int day = value(tk.nextToken(), 1, 31);
  int month = value(tk.nextToken(), 1, 12);
  int year = value(tk.nextToken(), 1900, 2100);
  if ((month == 4 || month == 6 || month == 9 || month == 11) && day == 31)
   throw new Exception("Illegal value");
  else if (month == 2 && day > (leapYear(year) ? 29 : 28))
   throw new Exception("Illegal value");
  this.date = String.format("%02d-%02d-%04d", day, month, year);
 }
 catch (Exception ex)
 {
  this.date = ex.toString();
 }
}

public void setTime(String time)
{
 try
 {
  StringTokenizer tk = new StringTokenizer(time, ":");
  if (tk.countTokens() != 3) throw new Exception("Illegal time");
  int tim = value(tk.nextToken(), 0, 23);
  int min = value(tk.nextToken(), 0, 59);
  int sek = value(tk.nextToken(), 0, 59);
  this.time = String.format("%02d:%02d:%02d", tim, min, sek);
 }
 catch (Exception ex)
 {
  this.time = ex.getMessage();
 }
}

public String toString()
{
 return date + " " + time;
}
```

```
 private int value(String text, int a, int b) throws Exception
 {
  int t = Integer.parseInt(text);
  if (a <= t && t <= b) return t;
  throw new Exception("Illegal value");
 }

 private boolean leapYear(int aar)
 {
  if (aar % 100 == 0) return aar % 400 == 0;
  return aar % 4 == 0;
 }
}
```

The example is chosen because I want to apply it later, but also to show that a bean is just an ordinary class, which can consist of everything that classes can. It is a bean that represents a date and time.

5 JSP

Above I have introduced servlets and shown how they can be used (and are used) to write dynamic websites, but it is also clear that it is a big and difficult task to develop complex web applications alone using servlets. Therefore JSP pages are introduced, which you can briefly characterize as documents that are a mixture of HTML and Java code. The Java code will then be executed on the server before the document is sent to the client's browser. This is best explained with an example.

I have in NetBeans created a new web application, which I have called *DateServer*, and the result is as before an application with a homepage called *index.html*. Next, I have right-clicked the Web Pages folder and selected *New* and then *JSP* and added a new JSP page, which I have called *start*, and the result is that a file named *start.jsp* is added to the project, whose content is almost an initial comment:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>JSP Page</title>
 </head>
 <body>
  <h1>Hello World!</h1>
 </body>
</html>
```

As you can see, it looks like a regular HTML document, and there are only two differences, which is the first line, and so the extension *jsp*. The latter is important as it tells the web server that the document is to be translated before it is sent to the browser. If you right-click on the name on the *Projects* tab and select *Run File*, the document will automatically be translated and opened in the browser.

Currently, there are no Java, but as the next step, I have right-clicked the *Source Packages* folder and added a class named *DateBean*. The code is the *Java bean*, as I have shown in the previous chapter. This Java bean can then be used from the JSP page as follows:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

```
  <title>JSP Page</title>
 </head>
 <body>
  <jsp:useBean id="dateBean" scope="page" class="dateserver.DateBean"/>
  <h1>Hello World!</h1>
  <h3>The data is: ${dateBean.date}</h3>
  <h3>The time is: ${dateBean.time}</h3>
 </body>
</html>
```

The code is expanded with a *jsp:bean* element, and as you see it, it defines a reference to the class *dateserver.DateBean* and calls this reference for *dateBean*. Finally, it defines a scope that tells where a *DateBean* object can be used. In addition, two lines are defined that insert the relevant bean values in the page. For example, inserts

```
${dateBean.date}
```

the property *date* in the HTML code. You should note the reference and you write *date* which is the property name, which means that it is the method *getDate()* that is being executed and that its return value is inserted. If you open the JSP page in the browser, the result is

and you can see that the bean class is instantiated and the object's values (the current date and time are inserted in the document). In particular, if you click on F5 (refresh of the browser), the browser will update the clock and then the object will be reinstated. You can use other public methods in a Java bean in the same way, but in this case there are no other. With regard of the value of the attribute scope there is four options:

1. *page* which is default and indicates that the bean object can be used solely from the current page
2. *request* indicating that the bean object can be used from all JSP pages that concerns the same request

3. *session* which means that the bean object can be used from all JSP pages regarding the same session – however, the page that creates the object must have a page directive with *session = "true"*
4. *application* indicating that the bean object can be used from all JSP pages in the application

When a request is received for a JSP page, the page is automatically translated into a servlet if it is not already. Here the web server will automatically check if the JSP page is newer than any servlet, and the page will be translated again. This means that a JSP page will only be translated into a servlet if needed.

You can also insert Java code into a JSP page using so-called scriplets. I have expanded the page in the following way:

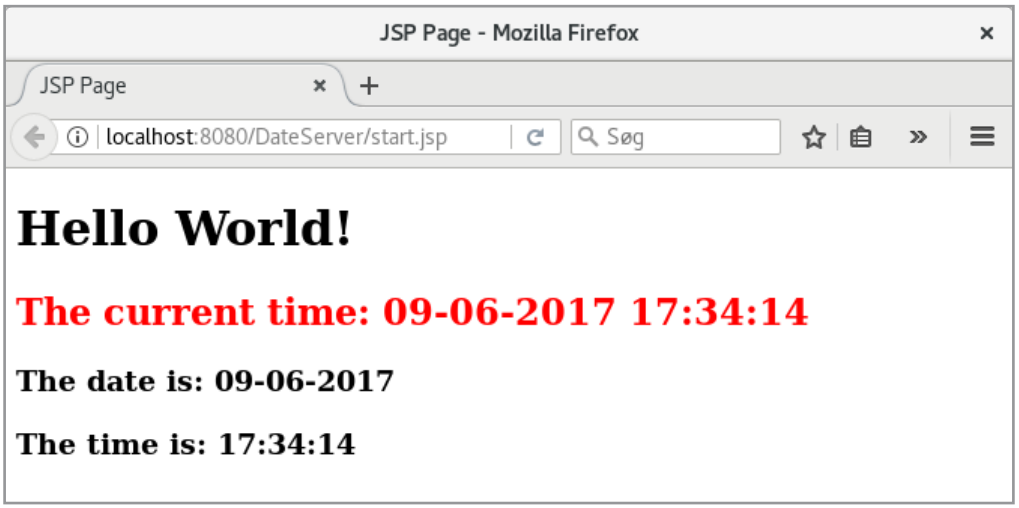
```
<%@page import="java.util.*"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<%! Calendar date = null;
 String text = null;
```

```
 String getTime(Calendar dt)
 {
  return String.format("%02d-%02d-%04d %02d:%02d:%02d", dt.get(Calendar.DATE),
   dt.get(Calendar.MONTH) + 1, dt.get(Calendar.YEAR),
   dt.get(Calendar.HOUR_OF_DAY), dt.get(Calendar.MINUTE),
   dt.get(Calendar.SECOND));
 }
%>
<% date = Calendar.getInstance();
  text = getTime(date);
%>
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>JSP Page</title>
 </head>
 <body>
  <jsp:useBean id="dateBean" scope="page" class="dateserver.DateBean"/>
  <h1>Hello World!</h1>
  <h2 style="color:red">The current time: <%=text%></h2>
  <h3>The date is: ${dateBean.date}</h3>
  <h3>The time is: ${dateBean.time}</h3>
 </body>
</html>
```

First, a page directive has been inserted to an *import* statement, which means that you can refer to the classes in the *java.util* package. Next, two scripting blocks are defined. The first starts with *<%!* and contains statements, which are variables and a method. It is code that is transmitted directly to the page's servlet and corresponds to defining instance variables and methods in a servlet. The second block starts with *<%* and contains code that is executed every time the page loads. In this case, the two variables defined in the first block are initialized. Finally, using the scriplets, you can insert values into the HTML code, such as

```
<%=text%>
```

that inserts the value of the variable *text*. The result of performing the page could then be as follows:



Including Java code in a JSP page that way (using scripting) has its uses, but is generally considered as poor programming. The reason is that it results in JSP pages that are very difficult to maintain. Therefore, it is recommended that you use Java beans as much as possible.

I will now make a last extension of the page:

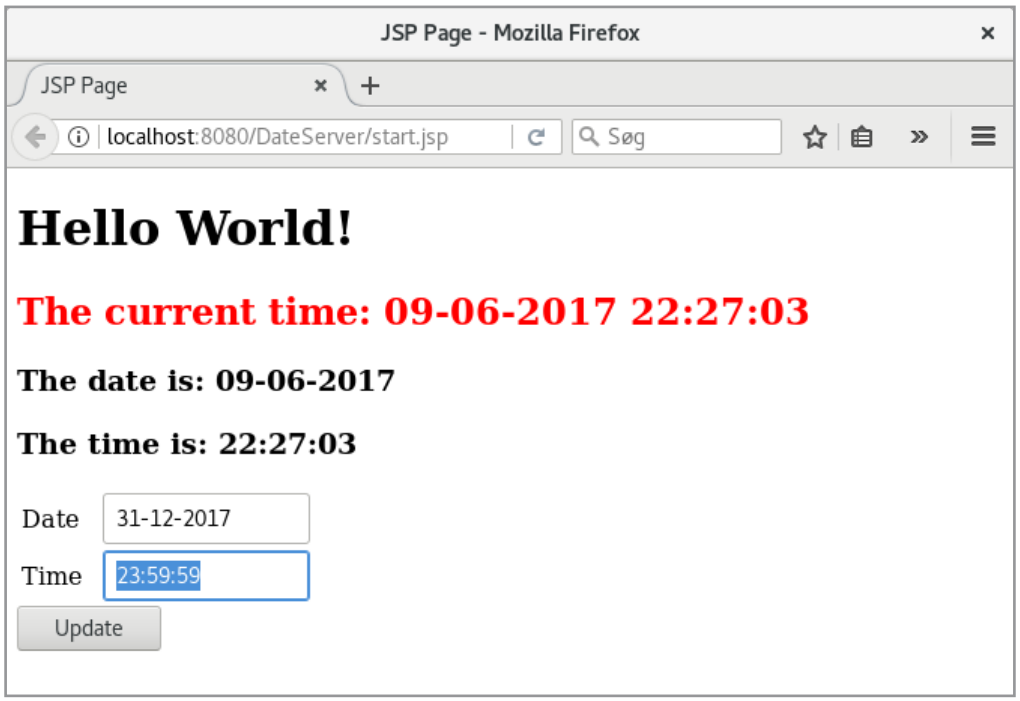
```
<%@page import="java.util.*"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<%! Calendar dato = null;
  String tekst = null;

  String getTime(Calendar dt)
  {
   return String.format("%02d-%02d-%04d %02d:%02d:%02d", dt.get(Calendar.DATE),
    dt.get(Calendar.MONTH) + 1, dt.get(Calendar.YEAR),
    dt.get(Calendar.HOUR_OF_DAY), dt.get(Calendar.MINUTE),
    dt.get(Calendar.SECOND));
  }
%>
<% dato = Calendar.getInstance();
   tekst = getTime(dato);
%>
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>JSP Page</title>
 </head>
 <body>
```

```
  <jsp:useBean id="dateBean" scope="page" class="dateserver.DateBean"/>
  <jsp:setProperty name="dateBean" property="*"/>
  <h1>Hello World!</h1>
  <h2 style="color:red">The current time: <%=tekst%></h2>
  <h3>The date is: ${dateBean.date}</h3>
  <h3>The time is: ${dateBean.time}</h3>
  <form method="post">
   <table>
    <tr>
     <td>Date</td>
     <td><input type="text" id="date" name="date" style="width:120px"/></td>
    </tr>
    <tr>
     <td>Time  </td>
     <td><input type="text" name="time" style="width:120px"/></td>
    </tr>
   </table>
   <input type="submit" value="Update"/>
  </form>
 </body>
</html>
```

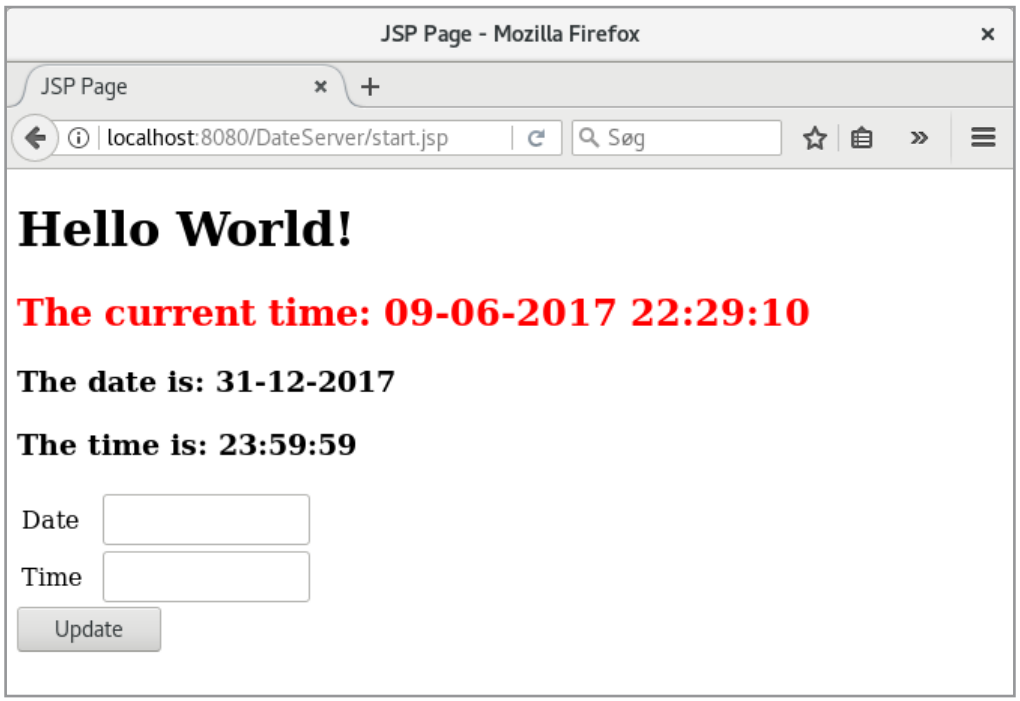
where a form has been added. Note that the form element does not define any action. This means that the page by a submit carries out a request to itself. The form defines two input components. Here it is important that they have the same name attributes that are the names of the two properties in *DateBean*. In addition, note the statement:

```
<jsp:setProperty name="dateBean" property="*"/>
```

It states that when performing a submit, all set methods in the class *DateBean* must be performed and updated with the values of the corresponding input fields. If you open the page, the result could be as shown below, where I have entered values in the two input fields:



If you click on the button then you get a window that shows that the current bean object is updated:



As can be seen from the above, it is simple on a JSP page to link input components to properties in a Java bean. Above it is all fields that are associated, but you can do it individually and for example. write

```
<jsp:setProperty name="dateBean" property="date"/>
```

if you do not want all fields to be updated. Another thing is handling exceptions, what I want to return to later, but in this case it is resolved in that way that the fields are assigned the exceptions in question, which of course is not a solution that can be used in practice.

As a last comment, a new web application starts creating a home page, which is a *HTML* document, and above it is not used for anything. For several reasons, it is advisable to start a web application with a HTML document, and in this case the home page could be modified to the following:

```
<!DOCTYPE html>
<html>
 <head>
  <title>TODO supply a title</title>
  <meta charset="UTF-8">
```

```
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script>
   window.location = "start.jsp";
  </script>
 </head>
 <body>
 </body>
</html>
```

As a result, the homepage sends the request to *start.jsp*. This is done using *JavaScript*, which is code executed on the client side, and you should just accept the syntax.

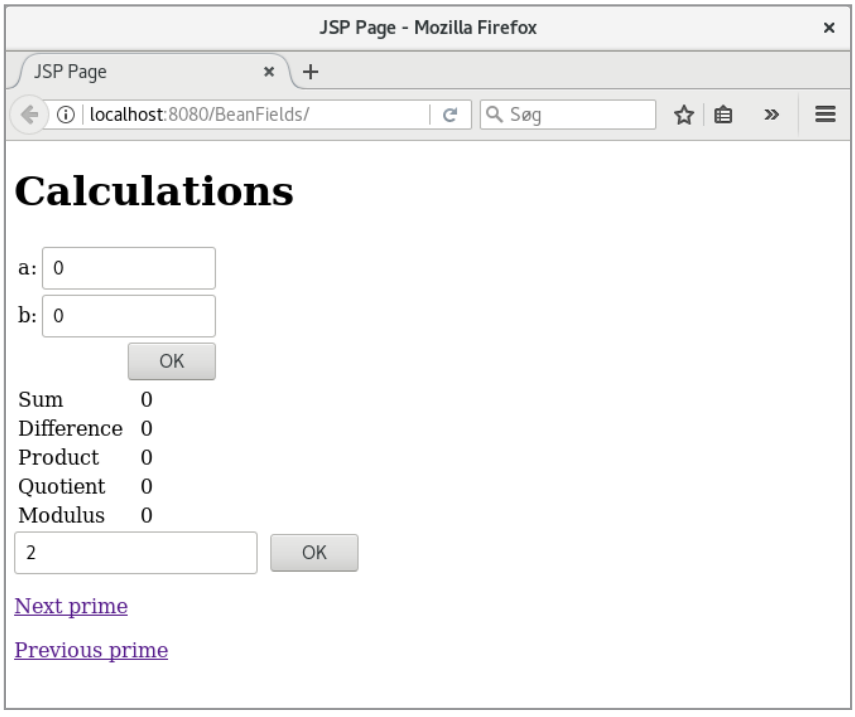
However, it is not a requirement that the home page is there, and if you wish, you can delete it and instead let the JSP page be the start page.

5.1 CALCULATIONS

In the introductory example of JSP, I have shown how simple it is to associate, for example, input fields in a JSP page with properties in a Java bean. It is also the theme in the following example (called *BeanFields*), but also I will show how, in a JSP page using other methods from a Java bean, and how to handle methods, including set-methods, which raises an exception. The project is again a web application project, but this time I have deleted the home page *index.html*. I have, however, add a JSP page named *index.jsp*. To get the application to use this page as the start page it may be required that you add a *web.xml* configuration file. If no, right-click the folder *WEB-INF* and select *Standard Deployment Descriptor (web.xml)*. The contents of the file must be as follows:

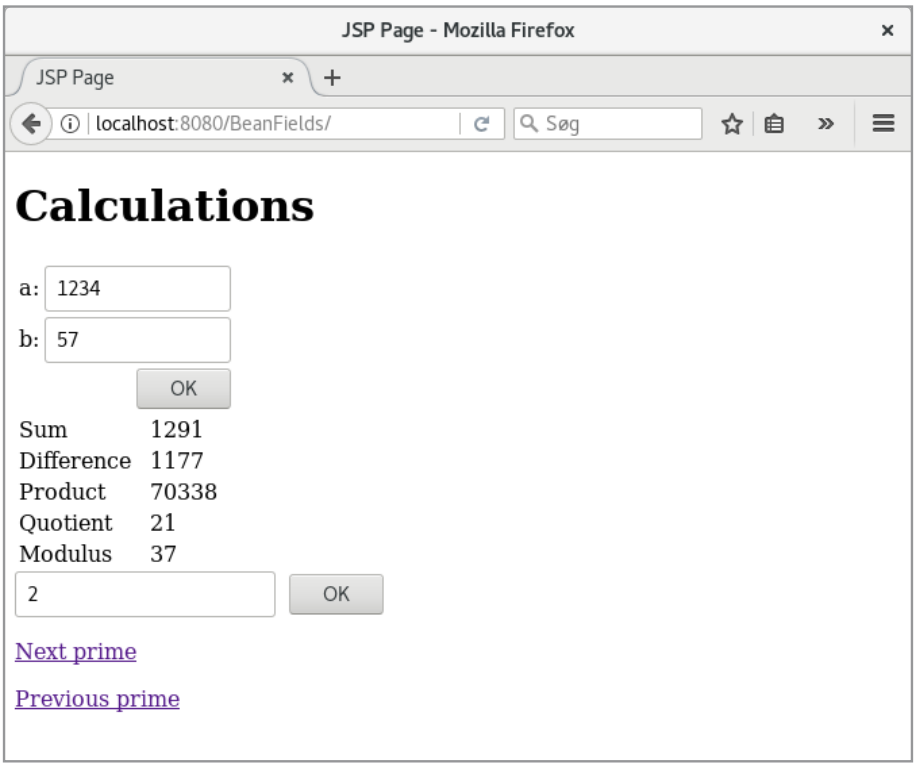
```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee" … >
  <welcome-file-list>
   <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <session-config>
   <session-timeout>
    30
   </session-timeout>
 </session-config>
</web-app>
```

When you opens the application in the browser, you get the window:



If you enter integers in the two top fields and click the top *OK* button, the result could be as shown below. That is, some calculations have been made on the two numbers, and the results appear on the five results lines. Then, in the bottom input field, enter 107 (which is a prime number) and click on the bottom *OK* button, nothing happens except that all fields retain their values, but you can see that the window is being updated and that there is a submit. If you now type 108 and click *OK*, you will see the window below. This is because 108 is not a prime and the application has raised an exception. When you click on *Repeat*, you return to the main window, and note that all values are still there. You should note in particular that the bottom input field still has the value 107. Finally, there are the two links. If you click on one of them, the lower input field will be updated by the next or the previous prime, respectively.

It was the functionality and the program should show how to use Java beans to separate the user interface HTML from the Java code, implemented in Java beans and other classes. One consider it as important, and you get a web application that is much easier to maintain.



The program uses two Java beans, which are located in the *beanfields.beans* package. The first represents the two top input fields and is the following:

```
package beanfields.beans;

import java.io.*;

public class MathBean implements Serializable
{
 private long arga;
 private long argb;

 public MathBean()
 {
 }

 public long getArga()
 {
  return arga;
 }
```

```
 public long getArgb()
 {
  return argb;
 }

 public void setArga(long arga)
 {
  this.arga = arga;
 }

 public void setArgb(long argb)
 {
  this.argb = argb;
 }

 public long add()
 {
  return arga + argb;
 }

 public long subtract()
 {
  return arga - argb;
 }

 public long multiply()
 {
  return arga * argb;
 }

 public long divide()
 {
  if (argb == 0) return 0;
  return arga / argb;
 }

 public long modulus()
 {
  if (argb == 0) return 0;
  return arga % argb;
 }
}
```

The class fills some lines, but is trivial. You should note that the class is written in accordance with the requirements of a Java bean, and here you should especially note the names of the two variables and the get and set methods that follows the standard of a Java bean. Also note that the class has other methods (the 5 calculation methods). Here you should especially note the two last ones who return 0 if the calculation is not possible. In fact, they should make an exception, but since the JSP page always uses the return value of the functions, this solution can not be used.

The other Java bean is the following:

```
package beanfields.beans;
import java.io.*;

public class PrimeBean implements Serializable
{
 private static final long max = 9223372036854775783L;
 private long prime = 2;

 public PrimeBean()
 {
 }

 public long getPrime()
 {
  return prime;
 }

 public void setPrime(long n) throws Exception
 {
  if (!isPrime(n)) throw new Exception("Illegal prime");
  prime = n;
 }

 public boolean next()
 {
  if (prime < max)
  {
   if (prime == 2) prime = 3;
   else for (prime += 2; !isPrime(prime); prime += 2);
   return true;
  }
  return false;
 }
```

```
 public boolean prev()
 {
  if (prime > 2)
  {
   if (prime == 3) prime = 2;
   else for (prime -= 2; !isPrime(prime); prime -= 2);
   return true;
  }
  return false;
 }

 public static boolean isPrime(long n)
 {
  if (n == 2 || n == 3 || n == 5 || n == 7) return true;
  if (n < 11 || n % 2 == 0) return false;
  for (long t = 3, m = (long)Math.sqrt(n) + 1; t <= m; t += 2)
   if (n % t == 0) return false;
  return true;
 }
}
```

Here there is also some code, and the goal is, among other things, to show that a Java bean can contain complex code similar to any other class. There is only one property that must represent a prime number. Therefore, its set method can raise an exception if you try to assign it a non-prime value. In addition to this property, there are two methods that assign the value to the next and the previous prime, respectively – if possible.

The program has another Java class, which is a servlet and is called *PrimesServlet*. The class is found in the package *beanfields.servlets* and *processRequest()* is:

```
@WebServlet(name = "Primes", urlPatterns = {"/Primes"})
public class PrimeServlet extends HttpServlet
{
 protected void processRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
 {
  PrimeBean bean = (PrimeBean)request.getSession().getAttribute("prim");
  String arg = request.getParameter("arg");
  if (arg.equals("next")) bean.next(); else bean.prev();
  response.sendRedirect("index.jsp");
 }
```

It is a very simple servlet. It starts by determining a reference to a session object, which is a *PrimeBean*, and then executes *next()* or *prev()* depending on the value of a parameter. Finally, a redirect is performed to the start page.

It was the Java code and back there are two JSP pages. The first is *index.jsp* where there are several things that you should notice:

```
<%@page contentType="text/html" session="true" pageEncoding="UTF-8"%>
<%@page errorPage="error.jsp" %>
<!DOCTYPE html>
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>JSP Page</title>
 </head>
 <body>
  <h1>Calculations</h1>
  <jsp:useBean id="math" scope="session" class="beanfields.beans.MathBean"/>
  <jsp:useBean id="prim" scope="session" class="beanfields.beans.PrimeBean"/>
  <jsp:setProperty name="math" property="*"/>
  <jsp:setProperty name="prim" property="prime"/>
  <form name="form1" method="post">
   <table>
```

```
    <tr>
     <td>a: </td>
     <td><input type="text" name="arga" value="${math.arga}"
      style="width: 120px"/></td>
    </tr>
    <tr>
     <td>b: </td>
     <td><input type="text" name="argb" value="${math.argb}"
      style="width: 120px"/></td>
    </tr>
    <tr>
     <td colspan="2" style="text-align: right">
     <input type="submit" value="OK"/></td>
    </tr>
   </table>
  </form>
  <table>
   <tr>
    <td>Sum</td><td>${math.add()}</td>
   </tr>
   <tr>
    <td>Difference  </td><td>${math.subtract()}</td>
   </tr>
   <tr>
    <td>Product</td><td>${math.multiply()}</td>
   </tr>
   <tr>
    <td>Quotient</td><td>${math.divide()}</td>
   </tr>
   <tr>
    <td>Modulus</td><td>${math.modulus()}</td>
   </tr>
  </table>
  <form name="form2" method="post">
   <input type="text" name="prime" value="${prim.prime}"/>  
   <input type="submit" value="OK"/>
  </form>
  <p><a href="Primes?arg=next">Next prime</a></p>
  <p><a href="Primes?arg=prev">Previous prime</a></p>
 </body>
</html>
```

The page starts with two directives. The first is added by NetBeans, but you should note that I have added an attribute:

```
session="true"
```

which means that any Java beans can be created as session objects. The next directive refers to a JSP error page and indicates a page that will automatically be redirected to in case of an exception. Here you can then make an appropriate error handling.

In the body section, the first thing is to create two Java bean objects. Here you should especially note that their scope is session, which means that they are objects whose value is retained in the context of submitting the page and it is an important part of the explanation that the page values are retained after a submit, however the actual reasoning is the two links at the end of the page. Following the declaration of the two beans, they are defined to be updated with the values of the input fields. That is, the classes' set methods must be performed. You should note that with * you can indicate that it is all properties, or you can specify exactly which property should be updated. This is the case with *prim*, although there is no reason for it in this case, since there is only this one property. You must then note how to tie a property in a bean to an input field:

```
<input type="text" name="arga" value="${math.arga}" style="width: 120px"/>
```

In particular, you should note how to refer to the individual property and to use the name of the variable alone. That's why a bean must adhere to the default for naming. Also note that the input field's *name* is the same name as the current property, what is required. Following the first form named *form1* follows a table that inserts the return values of the 5 calculation methods using the same syntax as when a property is linked to an input field. Note that a JSP page (and also an HTML page) must have more forms. When you click on a submit button, only the values for the components that are part of the same form as the button are submitted to the server. The other form has only a single input element that binds to a property for *prim*. This happens in the same way as explained above, but it's just another Java bean.

Finally, there are the two links that will send a request to a servlet. The servlet is called *Primes*, but an argument must be provided that tells which operation is to be performed. Here you should especially note the syntax for how to add an argument to an URL:

```
Primes?arg=next
```

Back there is only the error page, which is a quite simple JSP page:

```
<%@page contentType="text/html" pageEncoding="UTF-8" isErrorPage="true" %>
<!DOCTYPE html>
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>JSP Page</title>
 </head>
 <body>
  <h1>There has been an error</h1>
  <p><a href="index.jsp">Repeat</a></p>
 </body>
</html>
```

It is, of course, a very simple error handling, but it notifies the user that there has been an error rather than just letting the program go down.

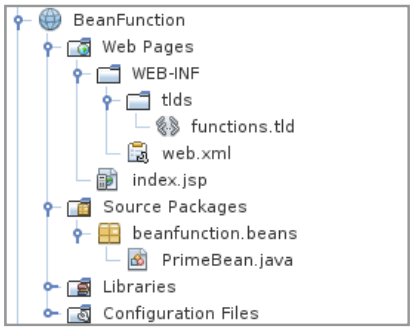
5.2 FUNCTIONS

It is also possible to call a static method in a class and apply the return value in the JSP page, but it requires the method to be registered in a *Tag Library Descriptor*, which is also called a *TLD*. The *BeanFunction* program opens the window above. There is a single input field that is bound to a property in a Java bean and there is a submit button. Clicking on the button sends the value of the field to the server and the program inserts the text *is not a prime* or *is a prime*. This happens by calling a static method whose argument is the value of the above property. The bean class code is the following and looks like the same bean from the previous example, but is just simpler:

```
package beanfunction.beans;

import java.io.*;

public class PrimeBean implements Serializable
{
 private long prime = 0;

 public PrimeBean()
 {
 }

 public long getPrime()
 {
  return prime;
 }

 public void setPrime(long n)
 {
  prime = n;
 }
```

```
 public static boolean isPrime(long n)
 {
  if (n == 2 || n == 3 || n == 5 || n == 7) return true;
  if (n < 11 || n % 2 == 0) return false;
  for (long t = 3, m = (long)Math.sqrt(n) + 1; t <= m; t += 2)
   if (n % t == 0) return false;
  return true;
 }
}
```

Note that the set method this time does not validate the value. To create a *Tag Library Descriptor*, I have added a directory named *tlds* to *WEB-INF* and then added the descriptor to this directory:



This sub directory is not necessary, but it is recommended to create such a directory as a larger web application may well have more *Tag Library Descriptors*. The descriptor itself is called *functions.tld* and is an XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd">
 <tlib-version>1.0</tlib-version>
 <short-name>f</short-name>
 <uri>/WEB-INF/tlds/functions</uri>
 <function>
  <name>isPrime</name>
  <function-class>beanfunction.beans.PrimeBean</function-class>
  <function-signature>boolean isPrime(long)</function-signature>
 </function>
</taglib>
```

In this case, the name is *functions.tld* and it has a so-called short-name called *f*. It is used to refer to a method. A single function has been defined, and the definition includes the function's name, the function's class and its signature. The document must contain a description for each static method that must be available for the JSP pages. Below is the code for the JSP page:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib uri="/WEB-INF/tlds/functions.tld" prefix="f" %>
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>JSP Page</title>
 </head>
 <body>
  <jsp:useBean id="prim" scope="request" class="beanfunction.beans.PrimeBean"/>
  <jsp:setProperty name="prim" property="prime"/>
  <h1>Prime number</h1>
  <form name="form1" method="post">
   <p>Enter a prime: <input type="text" name="prime" value="${prim.prime}"/></p>
   <br/>
   <input type="submit" value="OK"/>
  </form>
  <p>
   <c:choose>
```

```
    <c:when test="${f:isPrime(prim.prime)}">
     is a prime
    </c:when>
    <c:otherwise>
     is not a prime
    </c:otherwise>
   </c:choose>
  </p>
 </body>
</html>
```

First, note the definition of two TLDs:

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib uri="/WEB-INF/tlds/functions.tld" prefix="f" %>
```

A tag library descriptor expands JSP with new elements in the form of Java features, which can subsequently be used on the page. Here is the first a series of extensions that make it possible to introduce conditions and loops on the JSP page, while the latter is the descriptor defined above and which is part of the project. The two TLDs are referenced on the page using a prefix, which is *c* and *f*, respectively.

If you consider the body part, you can see that a bean is defined and its properties must be updated. Next, there is a form with an input field and a submit button, and compared to the previous example there is nothing new here. Next, a condition follows:

```
<c:choose>
 <c:when test="${f:isPrime(prim.prime)}">
  is a prime
 </c:when>
 <c:otherwise>
  is not a prime
 </c:otherwise>
</c:choose>
```

choose is a construct that is defined in the tag library, which is defined by the prefix *c*, and it corresponds to a switch statement. Each *when* element is controlled by a condition, and in this case, it is where the value returned from the bean object is a prime number. The method *isPrime()* is defined in the library that has the prefix *f* and you should note how to write the condition. You should also note how to state the argument for *isPrime()*. If the condition is met, the following HTML (here just text) is inserted into the document. Or, the subsequent *otherwise* element is performed. In this case, *choose* functions as an if/else, but there may be all the *when* elements that may be needed.

PROBLEM 1

You must solve the same task as problem 1 in the book Java 2, but this time it should be a web application. This means that you must write a loan calculation program, where the user must enter parameters for a loan (loan costs, loan amount, interest rate, number of years and number of periods per year). The program should then calculates the payment. The program should start with a form as shown below. If illegal values are entered:

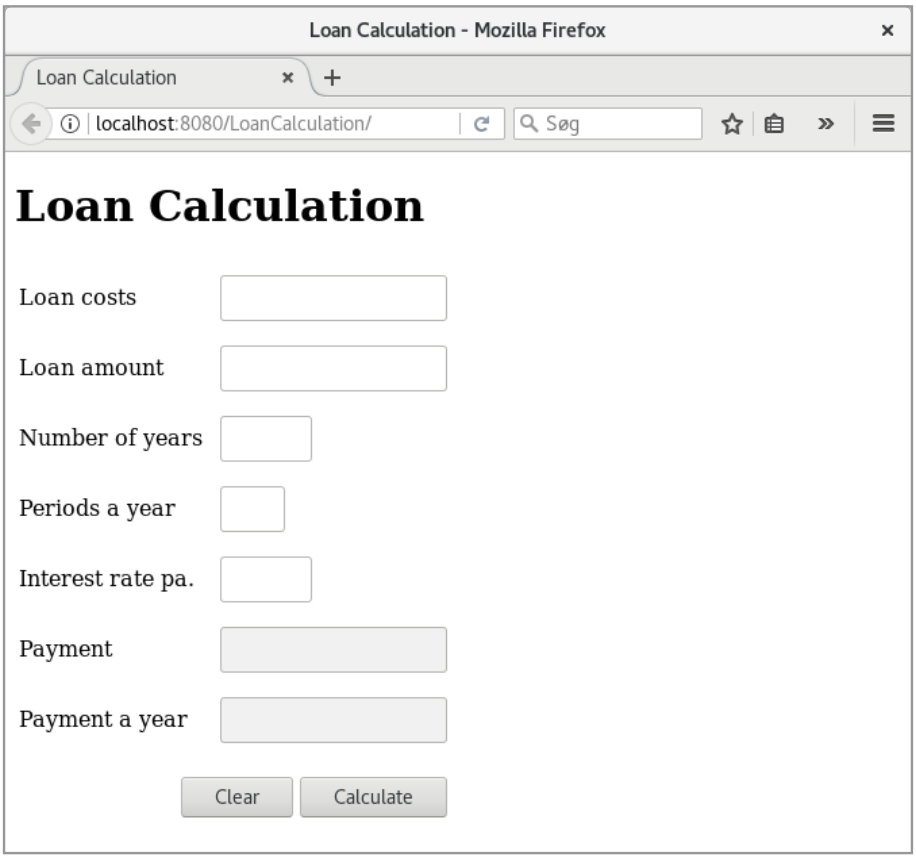
- the costs must not be negative
- the loan amount must not be negative
- number of years must be between 1 and 60
- number of periods a year must be between 1 and 12
- the interest rate must be greater than 0 and less than 25

you should instead be redirected to an error page.

If you click on the *Clear* button, the form fields must be blank.

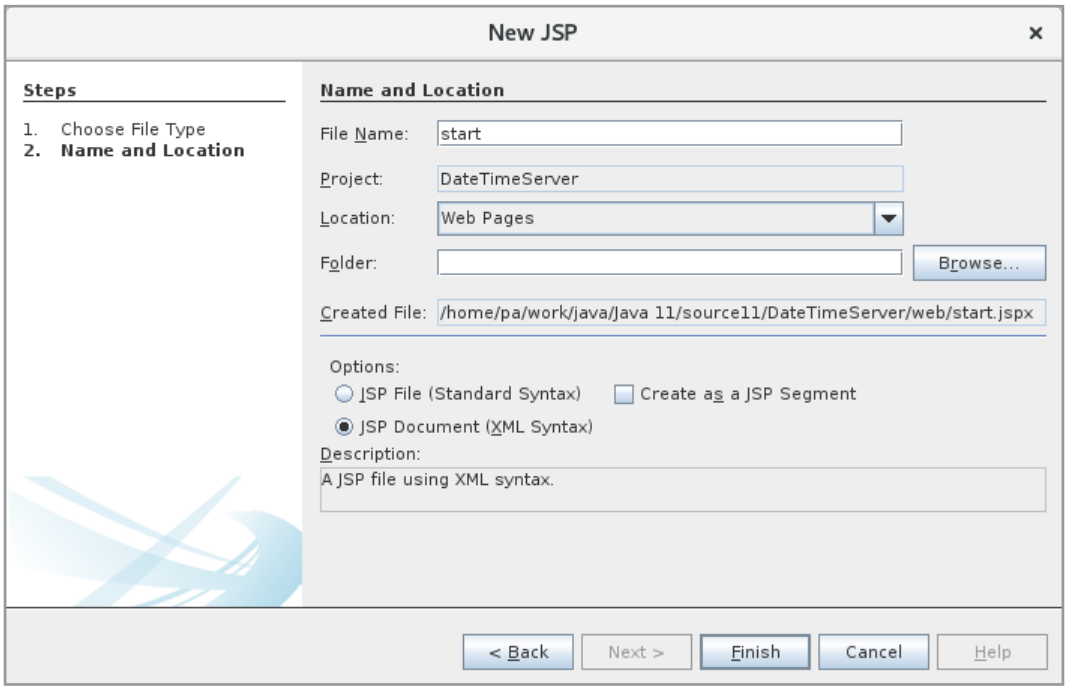
If legal loan parameters are entered and a loan calculation is made, an additional button must be displayed and clicking this button you should be redirected to a page showing an amortization plan.

The individual pages (home page and page with the amortization plan) must be written as JSP pages, and the loan should be represented by a Java bean.



5.3 JSP DOCUMENTS

When you create a jsp page in NetBeans you get the window as shown below, where for *Options* as default is selected *JSP File (Standard Syntax)*, and in most cases you will probably stick to it, but you can also select *JSP Document (XML Syntax)*. The result is a JSP page, which has the extension *jspx*, which resembles a standard JSP page, but where it is a requirement that the page is welformed XML. Since the above examples actually are already welformed, there will be no big difference besides being a requirement. Most importantly, however, a JSP document does not allow the use of scriptlets, thus forcing the programmer to place all Java code in Java beans and other classes, and that is the exact goal of this variation of JSP pages.



As an example, I will show the first program in this chapter (the web application *DateServer*), where the *start.jsp* page is replaced by a page *start.jspx*. The program is called *DateTimeServer* and performs exactly the same as the application *DateServer*. It uses the same Java bean *DateBean*, and additionally, the following simple bean is added:

```
package datetimeserver.beans;

import java.util.*;

public class CurrentTime implements java.io.Serializable
{
 public CurrentTime()
 {
 }

 public String getTime()
 {
  Calendar dt = Calendar.getInstance();
  return String.format("%02d-%02d-%04d %02d:%02d:%02d", dt.get(Calendar.DATE),
   dt.get(Calendar.MONTH) + 1, dt.get(Calendar.YEAR),
   dt.get(Calendar.HOUR_OF_DAY), dt.get(Calendar.MINUTE),
   dt.get(Calendar.SECOND));
 }
}
```

Then, the jspx page can be written as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
 <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"/>
 <jsp:element name="text">
  <jsp:attribute name="lang">EN</jsp:attribute>
  <jsp:body>
   <jsp:useBean id="dateBean" scope="page"
    class="datetimeserver.beans.DateBean"/>
   <jsp:useBean id="timeBean" scope="page"
    class="datetimeserver.beans.CurrentTime"/>
   <jsp:setProperty name="dateBean" property="*"/>
   <h1>Hello World!</h1>
   <h2 style="color:red">The current time: ${timeBean.time}</h2>
   <h3>The date is: ${dateBean.date}</h3>
   <h3>The time is: ${dateBean.time}</h3>
   <form method="post">
    <table>
     <tr>
      <td>Date</td>
      <td><input type="text" id="date" name="date" style="width:120px"/></td>
     </tr>
```

```
      <tr>
       <td>Time</td>
       <td><input type="text" name="time" style="width:120px"/></td>
      </tr>
     </table>
     <input type="submit" value="Update"/>
    </form>
  </jsp:body>
 </jsp:element>
</jsp:root>
```

As you can see, it's mostly copy and paste from the previous example, but you should notice three things. First of all, it is about wellformed XML. Secondly, there are no scriplets, and if you try to use the scriplets, you get an error. Third, there are several new elements of the form *jsp:xxxx*, and some may appear more than shown in this example.

As mentioned above, the aim of JSP documents is to force not to use scriplets. It's true that exaggerated use of scriplets leads to code that is harder to read and maintain, but conversely, it also has its uses, and generally I prefer standard JSP pages. First of all, with modern development tools like NetBeans, it's easy to make sure that the code is well-formed, and secondly, it means that because using of scriplets is allowed you do not have to use them or at least you can minimize the use and only use scriplets where it makes sense.

5.4 CHANGE ADDRESS 2

I've called this example for *ChangeAddress2*, because it's a bit the same problem as in the example *ChangeAddress1*, but it's really nothing about moving messages. The example shows a web application that will simulate that employees in a company can enter and maintain personal information such as name, address, email, and more. In addition to this, you should be able to get an overall overview of all employee data.

The exemple is slightly larger than the previous examples in this book, and it should show many of the solutions that are part of a typical web application. There are many details, but one of the main goals of the example is to show a web application that uses a database.

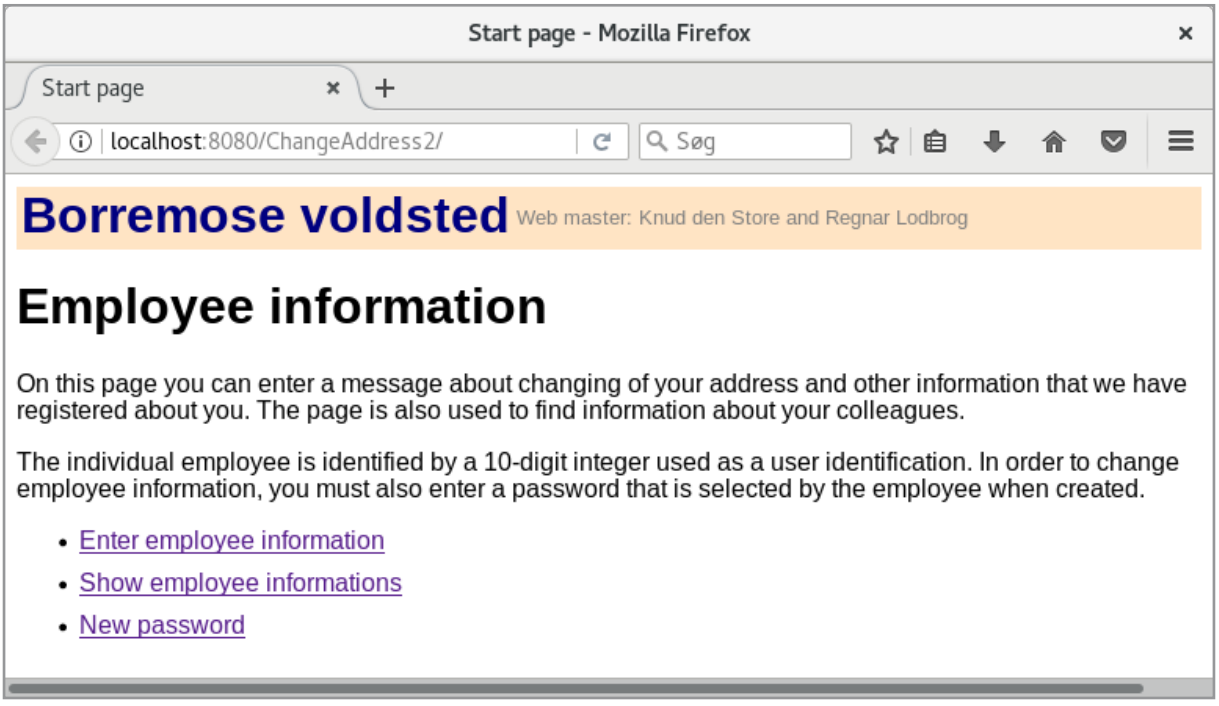
Therefore, I will start by describing the database, which use a table named *person* in database *padata*. In addition to this table, the table *zipcode* is used. The *person* table is defined as follows:

```
use padata;
drop table if exists persons;
```

```
create table persons
(
 id int not null auto_increment primary key,
 firstname varchar(40) not null,
 lastname varchar(30) not null,
 addrline1 varchar(30),
 addrline2 varchar(30),
 zipcode char(4) not null,
 phone varchar(20),
 email varchar(50) not null,
 title varchar(50),
 date date,
 passwd varchar(150),
 foreign key (zipcode) references zipcode(code)
);

alter table persons auto_increment=1001;
```

The meaning of the columns should appear from the names – perhaps the last two. The column *date* must contain the date when a row has last been updated, while the last column may contain a password, which should be encrypted. You should also note that in addition to the key are the only columns that must have a value *firstname*, *lastname*, *zipcode* and *email*, and note that the column *zipcode* is a foreign key to the table *zipcode*.

When you start the application, you get the following browser:



that can be perceived as a welcome page. It is a JSP page named *index.jsp*, and it has links to three other JSP pages, which are for the three functions that the application can perform. The page does not look much, but there is still a lot of new, because the page has a header (the colored area). It is a JSP page that is part of another JSP page. In addition, the application uses a style sheet and a bit of JavaScript, which are topics that are dealt with first in the next book.

Looking at the top line, it consists of a header as well as a text concerning the page's webmaster. The latter is an example of a custom web element. The example may in this case be a little searched, but a custom web element is defined as a derived class of the class *SimpleTagSupport*:

```
package changeaddress;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class Webmaster extends SimpleTagSupport
{
 private String master1 = null;
 private String master2 = null;
```

```
 public void setMaster1(String master)
 {
  master1 = master;
 }

 public void setMaster2(String master)
 {
  master2 = master;
 }

 @Override
 public void doTag() throws JspException
 {
  PageContext context = (PageContext) getJspContext();
  JspWriter out = context.getOut();
  try
  {
   out.println("Web master: " + master1);
   if(master2 != null)
   {
    out.println(" and " + master2);
   }
  }
  catch (Exception e)
  {
  }
 }
}
```

In this case, the element must have two attributes that are strings and are respectively called *master1* and *master2*. Both attributes have set methods, but otherwise the method *doTag()* defines how to render the item in the browser. In order for custom web elements to be used in the application, they must be registered in a *Tag Lidrary Description* file (which I previously used to define a function), and in this case, the content is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" xmlns="http://java.sun.com/xml/ns/javaee" … >
 <tlib-version>1.0</tlib-version>
 <short-name>t</short-name>
 <uri>/WEB-INF/tlds/taglib</uri>
 <tag>
  <name>webmaster</name>
  <tag-class>changeaddress.Webmaster</tag-class>
  <body-content>empty</body-content>
  <attribute>
```

```
    <name>master1</name>
    <rtexprvalue>true</rtexprvalue>
    <required>true</required>
   </attribute>
   <attribute>
    <name>master2</name>
    <rtexprvalue>true</rtexprvalue>
    <required>false</required>
   </attribute>
  </tag>
</taglib>
```

Note that, as a short-name, I have used the letter *t*. The file defines a single element called *webmaster*, and in addition to the name of the class, you must define the element's attributes, and in this case there are two. An attribute is defined by the name, and where it is required. The name *rtexprvalue* specifies whether the attribute must be initialized with a *${}* server value.

With this custom item in place, the header can be defined as the following JSP page, called *header.jsp*:

```
<%@taglib uri="/WEB-INF/tlds/taglib.tld" prefix="t" %>
<html xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
 <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"/>
 <div class="header-class" style="height: 40px;">
  <table>
   <tr>
    <td style="font-size: xx-large; font-weight: bold; color: navy;">
     Borremose voldsted</td>
    <td style="font-size: small; color: gray;">
     <t:webmaster master1="Knud den Store" master2="Regnar Lodbrog"/></td>
   </tr>
  </table>
  <br/>
 </div>
</html>
```

You should note that the tag library described above is defined by a directive, but otherwise you should notice that there is no body. It's not a real JSP page, but some JSP that is intended to be inserted into another JSP page.

One of the major challenges in web application development is the visual repræsentation and thus how the page is displayed in the browser. This is something that I will only touch on to a limited extent, but typically you define the details with styles. That is for each of the two TD elements above, there is defined a style that indicates font size and color. More generally, you can define styles in a style sheet, which is a document of styles that the application's pages can apply. As part of this application, I have defined the following stylesheet:

```
body
{
 font-family: "Liberation Sans"
}

p
{
 width: 800px
}

li
{
 margin: 10px 0
}
```

```
.error
{
 font-weight: bold;
 color: red
}

.header-class
{
 background: bisque
}

table.imagetable
{
 font-family: verdana,arial,sans-serif;
 font-size:11px;
 color:#333333;
 border-width: 1px;
 border-color: #999999;
 border-collapse: collapse;
}

table.imagetable th
{
 background:#b5cfd2 url('cell-blue.jpg');
 border-width: 1px;
 padding: 8px;
 border-style: solid;
 border-color: #999999;
}

table.imagetable td
{
 background:#dcddc0 url('cell-grey.jpg');
 border-width: 1px;
 padding: 8px;
 border-style: solid;
 border-color: #999999;
}
```

There are, of course, many syntactical details related to style sheets, but when you read the individual styles, they are easy enough to understand, and NetBeans knows style heets and also provides assistance to the individual styles. In practice, a style sheet can be very comprehensive and the whole idea is to move the definition of how the individual HTML elements should be displayed from the HTML and JSP documents. This means, on the one hand, that these documents become simpler and easier to understand, and partly that it is easier to maintain how websites appear, as you only has to modify the style sheet. The idea is that the HTML and JSP documents should only define the content, while style sheets defines how the content is rendered in the browser. A style sheet is thus a text document sent with the page, as the browser uses to render the document. A style sheet is not used on the server side.

In this case, 9 styles are defined. The first three is very simple and defines the font for the body element (and thus the entire document) that as default should apply, and how wide a paragraph should be (such a text line does not necessarily fill the entire screen), and that a *li* element must have an upper margin of 10. The next two defines a so-called *class* that can be associated with specific elements. For example *header-class* that is used in the above header:

```
<div class="header-class">
```

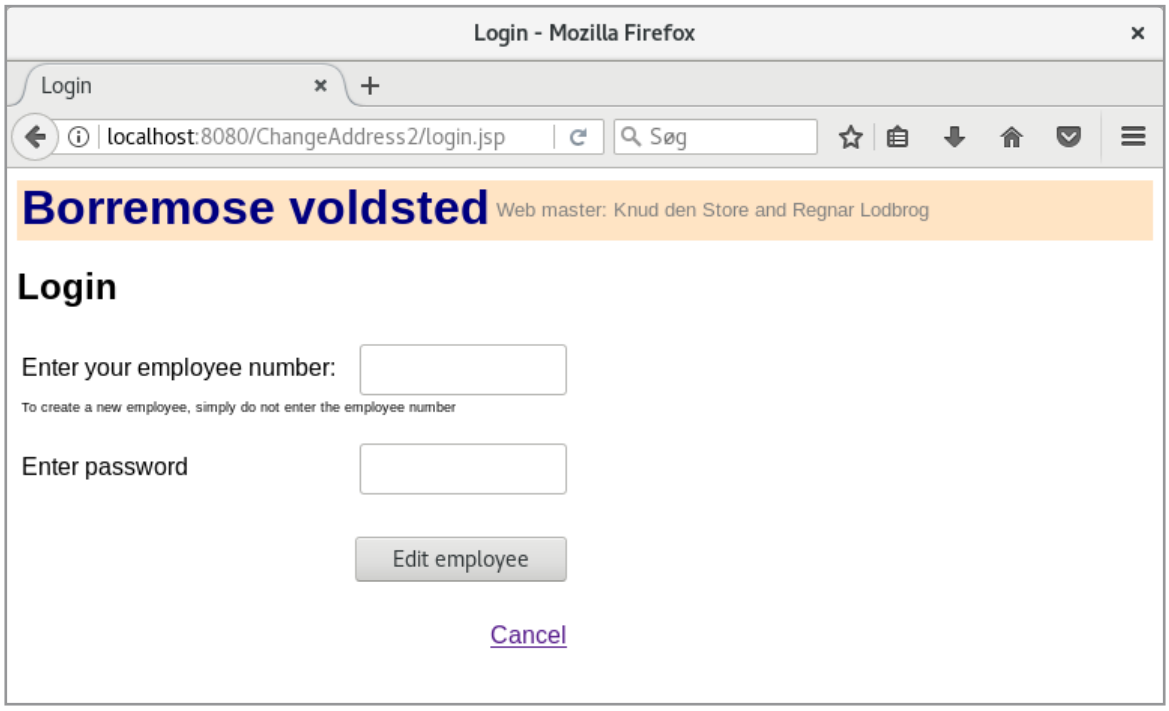
The last three styles are more complex and define styles that can be used in a HTML table. I will return to them later. These styles are used by all documents that uses that style sheet. If you look at the code for the home page, you can see how how with a *link* element in the header yoy defines to use the style sheet in question:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Start page</title>
  <link rel="stylesheet" href="styles.css" type="text/css"/>
 </head>
 <body>
  <jsp:include page="header.jsp"/>
  <h1>Employee information</h1>
  <p>On this page ….. </p>
  <p>The individual employee ….. </p>
  <ul>
```

```
    <li><a href="login.jsp">Enter employee information</a></li>
    <li><a href="show.jsp">Show employee informations</a></li>
    <li><a href="passwd.jsp">New password</a></li>
  </ul>
 </body>
</html>
```

You should also note how the above header is inserted with a *jsp:include* element as the first line in the body section. You must note that it is on the server side and it is the server that is responsible for creating the completed document. In this application, all pages start with this header and it can help ensure that all pages in a web application get a common look and feel and it is easy to change as you can easily change the header. You should also note that, in principle, a JSP page may have all of the *jsp:include* items that may be desired and they may appear anywhere.

If you click on the top link from the home page, you get the following browser:



It should illustrate a simple login page. In this case, the user must enter his employee number and password and when the button is clicked, this information will be sent to the server. If the user only enter a password (there is no employee number), it is perceived as creating a new employee, and otherwise it means that you can edit the employee information. For practical use, it may not be appropriate, but I do not want to make the application more extensive than necessary. The page is of course basically a form and, in principle, it is a very simple page, which simply consists in sending a form to the server, but there are nevertheless one having challenges to be resolved.

When the form is sent to the server, the server must validate if the entered data is legal. In this case, the employee number must be a 4-digit integer (automatically assigned by the database server) and there must be a password. If the employee number is blank, the server must create a new employee and assign it to an auto generated number. If an employee number has been entered, the server must validate the number and, if it is legal, the server must check if the database has an employee with that number and password and, if necessary, load the employee's data. If the form's data is illegal, the program must return to the login with an error message and the form fields must retains there content so that the user can edit the content. If data can be accepted, the server instead has to go to the editing page so that the user can edit the data in question.

All that requires some Java code, and I want to start with a simple bean that can represent data to the login page:

```
package changeaddress.beans;

import java.io.*;

public class LoginBean implements Serializable
{
 private String userid;
 private String passwd;
 private String error;

 public LoginBean()
 {
 }

 public String getUserid()
 {
  return userid;
 }

 public void setUserid(String userid)
 {
  this.userid = userid;
 }

 public String getPasswd()
 {
  return passwd;
 }

 public void setPasswd(String passwd)
 {
  this.passwd = passwd;
 }

 public String getError()
 {
  return error;
 }

 public void setError(String error)
 {
  this.error = error;
 }
}
```

There is not much to explain, but you should notice that there is an additional property that I have called *error* and it is used to send an error message back. Then, the JSP page can be written as follows:

```
<%@page contentType="text/html" pageEncoding="UTF-8" session="true"%>
<%@page errorPage="error.jsp" %>
<!DOCTYPE html>
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Login</title>
  <link rel="stylesheet" href="styles.css" type="text/css"/>
  <script src='scripts.js'></script>
  <script>
   function encryptPassword()
   {
    var passwd = ("" + document.getElementById("passwd").value).trim();
    if (passwd.length > 0)
     document.getElementById("passwd").value = createHash(passwd);
    return true;
   }
  </script>
 </head>
```

```
 <body>
  <jsp:useBean id="loginBean" class="changeaddress.beans.LoginBean"
   scope="session"/>
  <jsp:include page="header.jsp"/>
  <h2>Login</h2>
  <form method="post" action="PersonServlet" onsubmit="return encryptPassword()">
   <table>
    <tr>
     <td>Enter your employee number:&amp;nbsp;&amp;nbsp;&amp;nbsp;</td>
     <td><input type="text" id="mednr" name="userid" style="width:120px"
      value="${loginBean.userid}"/></td>
    </tr>
    <tr>
     <td colspan="2"><div style="font-size: xx-small">
      To create a new employee, simply do not enter the employee number
      </div></td>
    </tr>
    <tr>
     <td colspan="2" style="height: 10px"></td>
    </tr>
    <tr>
     <td>Enter password</td>
     <td><input type="password" id="passwd" name="passwd" style="width:120px"
      value="${loginBean.passwd}"/></td>
    </tr>
    <tr>
     <td colspan="2" style="height: 20px"></td>
    </tr>
    <tr>
     <td colspan="2" style="text-align: right"><input type="submit" id="edit"
      name="edit" value="Edit employee"/></td>
    </tr>
    <tr>
     <td colspan="2" style="height: 20px"></td>
    </tr>
    <tr>
     <td colspan="2" style="text-align: right"><a href="start">Cancel</a></td>
    </tr>
   </table>
  </form>
  <div class="error">${loginBean.error}</div>
 </body>
</html>
```

Note that line 2 refers to an error page. In principle, you should never come to this page because the server validates data, but you will be sent to the error page in case of one or more serious errors, and that is, when an exception occurs. Also note how the above *LoginBean* is applied and how its properties are linked to the input fields as well as to a *div* element at the end of the document. It is used to show a possible error message. Please note that the scope of the particular bean is a session. That is that the value of objects is preserved throughout the session. Finally, you should note that the page only reads the bean object, but does not update it. Otherwise, you may notice that the page uses the style sheet, and that one input field has the type password. It simply means that the user's entries are not displayed as text.

However, the most important thing is the form element, where its action indicates that the form should be sent to *PersonServlet*, which is the Java object to process the form, and finally the form element states that before submitting, the JavaScript function *encryptPassword()* must be performed. The user's password must be stored encrypted in the database, and I have previously shown (in the book Java 7) how to do it. Java has classes for it, which determines a message digest of the password and stores it in the database. However, it causes a problem as it can only happens on the server side, and the password must therefore be sent to the server in clear text, allowing others to intercept it. The encryption must therefore be done on the client side, and here I currently only have JavaScript available. A message digest is determined by means of a complex algorithm, and there are several that can be used and which differ in terms of their strength and thus how difficult they are to break. The algorithm is well described on the Internet, and you can easily find an example where it is implemented in JavaScript. The file *scripts.js* implements such an algorithm (SHA-1 algorithm). It is not the strongest algorithm, but for many practical purposes strong enough. I do not want to review the algorithm here or show the code but briefly the idea is to modify the text in such a way that it can not be recognized and that you can not regret and from that message digest determine the password. However, there is a problem as it happens on the client side, so everyone can see what's happening and therefore it is discussed whether it's safe to encrypt a password using script code, because such a code is not irreproachable, but in relation to this example, it must be sufficient. Should encryption be safe, one will usually require another approach (another implementation) than what this example shows.

Before I look at *PersonServlet*, I will look at the class *PersonBean*, which is a Java bean that defines a person. The class is extensive and I have only shown a part of the code:

```
public class PersonBean implements Serializable
{
 private static ArrayList<Zipcode> list = new ArrayList();
 private int userid;
 private String firstname;
```

```
private String lastname;
private String addrline1;
private String addrline2;
private String code;
private String city;
private String phone;
private String email;
private String title;
private Calendar date;
private String passwd;
private String error;

public PersonBean()
{
}

public void setUserid(String userid) throws Exception
{
 if (userid.trim().length() == 0)
 {
  this.userid = 0;
  return;
 }
```

```
 try
 {
  int id = Integer.parseInt(userid);
  if (id < 1000 || id > 9999) throw new Exception(" … ");
  this.userid = id;
 }
 catch (Exception ex)
 {
  throw new Exception("Illegal value for userid: " + ex.getMessage());
 }
}

public void setFirstname(String firstname) throws Exception
{
 firstname = firstname.trim();
 if (firstname.length() == 0) throw new Exception(" … ");
 this.firstname = Tools.cut(firstname, 40);
}

public void setCode(String code) throws Exception
{
 code = code.trim();
 for (Zipcode zc : list)
  if (zc.code.equals(code))
  {
   this.code = zc.code;
   this.city = zc.city;
   return;
  }
 throw new Exception("Illegal zipcode");
}

public String getDate()
{
 if (date == null) return "";
 return String.format("%02d-%02d-%04d", date.get(Calendar.DATE),
  date.get(Calendar.MONTH) + 1, date.get(Calendar.YEAR));
}

public void setDate(String date) throws Exception
{
 try
 {
  java.text.DateFormat formatter =
   new java.text.SimpleDateFormat("dd-MM-yyyy");
  if (this.date == null) this.date = java.util.Calendar.getInstance();
```

```
    this.date.setTime(formatter.parse(date));
   }
   catch (Exception ex)
   {
    throw new Exception("Illegal date");
   }
  }

  static
  {
   try (Connection conn = DB.getConnection();
    Statement stmt = conn.createStatement())
   {
    ResultSet res = stmt.executeQuery("SELECT * FROM zipcode");
    while (res.next())
     list.add(new Zipcode(res.getString("code"), res.getString("city")));
   }
   catch(Exception ex)
   {
   }
  }
}

class Zipcode
{
 public String code;
 public String city;

 public Zipcode(String code, String city)
 {
  this.code = code;
  this.city = city;
 }
}
```

Looking at the class's variables, it is clear that it reflects a row in the database and thus can represent a person. However, the class further defines a few variables as *city* and *error*. Here, the last one should be used for an error message when editing a person's data. Similar to the database definition, the following fields must be validated:

- useris
- firstname
- lastname
- code
- email

The *set* methods for these variables raise an exception in case of errors. For example the method *setFirstname()* that ensures that a first name has a value. The variable *email* is validated by a method *isMail()*, which is found in the class *Tools*. It's the same method I've shown before. Then there is finally the variable *code*, and since the corresponding column in the database is a foreign key, the method *setCode()* must test if the zip code is found. In order, that the method does not always have to connect to the database and perform a SELECT, the zip codes are loaded once for all in an *ArrayList<Zipcode>*. Here's *Zipcode* is a simple class that can represent a zip code by two fields. Reading the zip codes takes place in a static constructor, and in principle it does not require any special explanation. However, it requires a database connection and the class *DB* has a method that can create this connection to the database. I will wait to describe this class for later, but it will be used in several places. Once the arraylist is initialized, it is easy for the method *setCode()* to validate if a zip code is legal. If necessary, it also initializes the variable *city*.

Then there is *PersonServlet*, that is the servlet that JSP page *login.jsp* posts the form to when clicking on the submit button:

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

```
protected void processRequest(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException
{
 request.setCharacterEncoding("UTF-8");
 LoginBean bean = (LoginBean)request.getSession(true).getAttribute("loginBean");
 String usr = request.getParameter("userid").trim();
 String pwd = request.getParameter("passwd").trim();
 if (pwd.length() == 0)
 {
  bean.setError("You must enter password");
  bean.setUserid(usr);
  bean.setPasswd("");
  response.sendRedirect("login.jsp");
  return;
 }
 PersonBean person = new PersonBean();
 if (usr.length() > 0)
 {
  if (!userOk(usr))
  {
   bean.setError("Illegal user ID");
   bean.setUserid(usr);
   bean.setPasswd("");
   response.sendRedirect("login.jsp");
   return;
  }
  if (!loadPerson(person, usr, pwd))
  {
   bean.setError("Employee number not found or illegal password");
   bean.setUserid(usr);
   bean.setPasswd("");
   response.sendRedirect("login.jsp");
   return;
  }
 }
 else person.setPasswd(pwd);
 request.getSession().setAttribute("personBean", person);
 request.getSession().removeAttribute("loginBean");
 request.getRequestDispatcher("/edit.jsp").forward(request, response);
}

private boolean userOk(String userid)
{
 try
 {
  int t = Integer.parseInt(userid);
  return t > 1000 && t < 10000;
 }
```

```
  catch (Exception ex)
  {
   return false;
  }
 }

 private boolean loadPerson(PersonBean person, String userid, String passwd)
 {
  try (Connection conn = DB.getConnection();
   Statement stmt = conn.createStatement())

  {
   ResultSet res = stmt.executeQuery("SELECT * FROM persons WHERE id = " +
    userid + " AND passwd = '" + passwd + "'");
   if (res.next())
   {
    person.setUserid("" + res.getInt("id"));
    person.setFirstname(res.getString("firstname"));
    person.setLastname(res.getString("lastname"));
    person.setAddrline1(res.getString("addrline1"));
    person.setAddrline2(res.getString("addrline2"));
    person.setCode(res.getString("zipcode"));
    person.setPhone(res.getString("phone"));
    person.setEmail(res.getString("email"));
    person.setTitle(res.getString("title"));
    person.setDate(Tools.toStr(res.getDate("date")));
    return true;
   }
   return false;
  }
  catch(Exception ex)
  {
   return false;
  }
 }
}
```

I have only shown that part of the code that explains what happens and it is primarily *processRequest()*. First note the line

```
request.setCharacterEncoding("UTF-8");
```

It is necessary to ensure that Danish letters are processed correctly when the form is submitted. Perhaps it is not so important in this example, but in connection with the next form it is. The first thing that happens is to add a reference to the bean object:

```
LoginBean login = (LoginBean)request.getSession().getAttribute("loginBean");
```

Note that the object exists as it is defined with session scope. Next, the value of the employee number and password (which is now encrypted) are determined and the method tests whether a password has been submitted. If not, an error message is attached to the bean object, and the server performs a redirect back to *login.jsp*. Since the bean object still exists, the page will display the error message. Has a password been entered a *PersonBean* object is created. If an employee number has been entered (the length is positive) is validated first, if it is a positive 4-digit integer. If not, an error message will be returned in the same way as above. Otherwise, the method *loadPerson()* is called to try to read a person from the database with the appropriate employee number and password. Is it possible, *PersonBean* initializes the object with the entire data, but otherwise, again, returns to *login.jsp* with an error message. If, on the other hand, the entered employee number is blank, *PersonBean* initializes the object with the entered password.

If password and employee number can be accessed, then *PersonBean* is saved as a session object, while at the same time removing the session object *loginBean*. It is necessary to ensure that the item does not live next time the login page opens. Finally, a forward is made to the page *edit.jsp*, where the *PersonBean* object can be edited.

As a last comment to *login.jsp* there is a link to cancel. If you click on that link, you have to get back to the home page, but instead of doing it directly, a servlet *StartServlet* is called. Its *processRequest()* is the following:

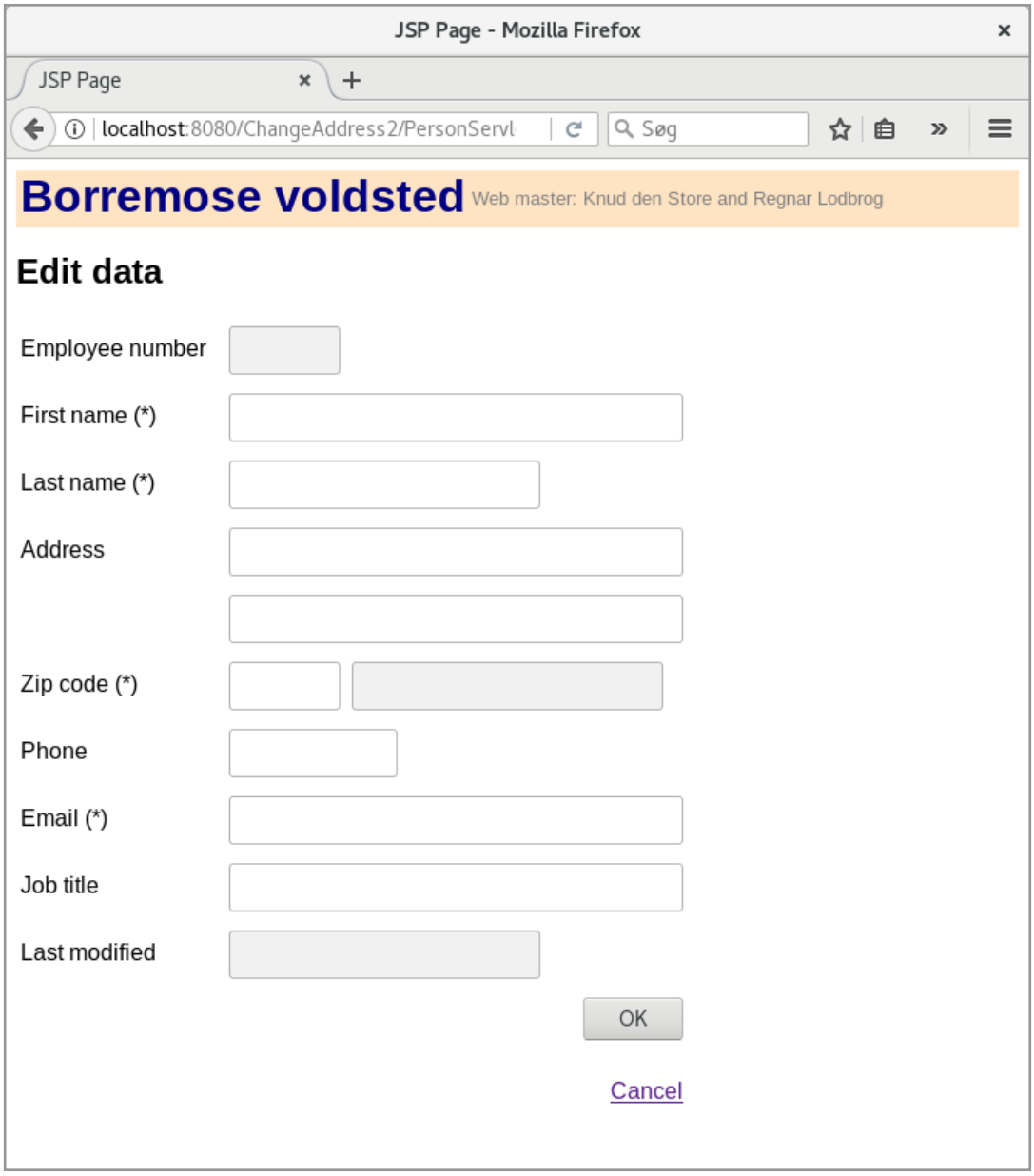
```
@WebServlet(name = "start", urlPatterns = {"/start"})
public class StartServlet extends HttpServlet
{
 protected void processRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException
 {
  HttpSession session = request.getSession(false);
  if (session != null) session.invalidate();
  request.getRequestDispatcher("/index.jsp").forward(request, response);
 }
```

The goal is to end the session so that the object *loginBean* (and, if any, other session objects) no longer lives.

Then there is *edit.jsp*, in the event of a new employee opens the page below. The page is basically a form and is in principle no different than *login.jsp*. It fills a lot, and I've only shown a part of the HTML code:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page errorPage="error.jsp" %>
<!DOCTYPE html>
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>JSP Page</title>
  <link rel="stylesheet" href="styles.css" type="text/css"/>
 </head>
 <body>
  <jsp:useBean id="personBean" class="changeaddress.beans.PersonBean"
   scope="session"/>
  <jsp:include page="header.jsp"/>
  <h2>Edit data</h2>
  <form method="post" action="StoreServlet">
   <table>
    <tr>
     <td>Employee number   </td>
     <td><input type="text" name="userid" readonly="true" style="width:60px"
      value="${personBean.userid}" /></td>
    </tr>
    …
    <tr>
```

```
      <td colspan="2" style="text-align: right"><a href="start">Cancel</a></td>
     </tr>
    </table>
   </form>
   <br/><br/>
   <div class="error">${personBean.error}</div>
  </body>
</html>
```



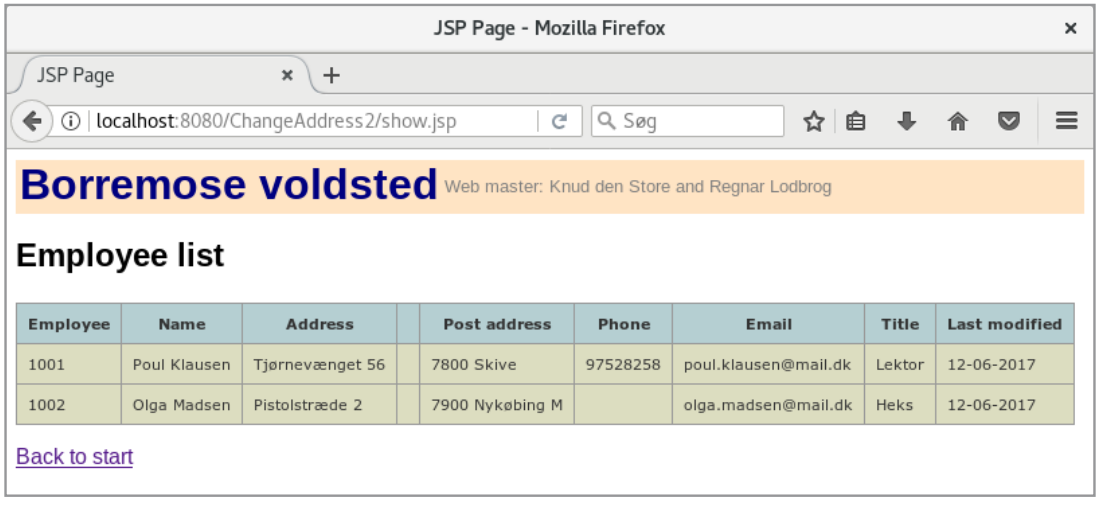
Compared to *login.jsp* there is nothing new to explain and it works in principle in the same way. You should note that there is a reference to the object *personBean*, and that when submitting the form, a request is made to a servlet named *StoreServlet*. It's *processRequest()* is the following:

```
protected void processRequest(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException
{
 response.setContentType("text/html;charset=UTF-8");
 PersonBean person = (PersonBean)request.getSession().getAttribute("personBean");
 try
 {
  request.setCharacterEncoding("UTF-8");
  person.setFirstname(request.getParameter("firstname"));
  person.setLastname(request.getParameter("lastname"));
  person.setAddrline1(request.getParameter("addrline1"));
  person.setAddrline2(request.getParameter("addrline2"));
  person.setCode(request.getParameter("code"));
  person.setPhone(request.getParameter("phone"));
  person.setEmail(request.getParameter("email"));
  person.setTitle(request.getParameter("title"));
  if (person.getDate().length() == 0)
  {
```

```
    person.setUserid("" + create(person));
    request.getRequestDispatcher("/employee.jsp").forward(request, response);
   }
   else
   {
    update(person);
    request.getRequestDispatcher("/start").forward(request, response);
   }
  }
  catch (Exception ex)
  {
   person.setError(ex.toString());
   request.getRequestDispatcher("/edit.jsp").forward(request, response);
  }
}
```

It starts by initializing the session object with the form data. Several of the set methods can raise an exception if the value is illegal and if it happens, the server performs a forward back to *edit.jsp* with an error message. Initializing the object correctly, the *date* field is used to know whether it is an update or to add a new row to the database. This happens by using two methods that I have not shown here, but the difference is that one executes a SQL INSERT, while the other performs a SQL UPDATE. If these methods are not performed properly, they raise an exception, which is sent back to *edit.jsp* as an error message. If, on the other hand, the database is updated correctly, you are in case of an update send back to the home page, but if it is a new employee, you will be sent to a page *employee.jsp*, which shows which employee number you are assigned.

If you click on the center link on the start page, you will get an overview of all employees in the database. The result could, for example be, as shown below, where two employees have been created:



The table's look and feel is determined by the style sheet, where the last three classes are used to style the table. The JSP page content are as follows:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page errorPage="error.jsp" %>
<!DOCTYPE html>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>JSP Page</title>
  <link rel="stylesheet" href="styles.css" type="text/css"/>
 </head>
 <body>
  <jsp:useBean id="personsBean" class="changeaddress.beans.PersonsBean"
   scope="page"/>
  <jsp:include page="header.jsp"/>
  <h2>Employee list</h2>
  <table class="imagetable">
   <tr>
    <th>Employee</th>
    <th>Name</th>
    <th>Address</th>
    <th></th>
    <th>Post address</th>
    <th>Phone</th>
    <th>Email</th>
    <th>Title</th>
    <th>Last modified</th>
   </tr>
   <c:forEach items="${personsBean.persons }" var="pers">
   <tr>
    <td>${pers.userid}</td>
    <td>${pers.firstname} ${pers.lastname}</td>
    <td>${pers.addrline1}</td>
    <td>${pers.addrline2}</td>
    <td>${pers.code} ${pers.city}</td>
    <td>${pers.phone}</td>
    <td>${pers.email}</td>
    <td>${pers.title}</td>
    <td>${pers.date}</td>
   </tr>
   </c:forEach>
  </table>
  <p><a href="start">Back to start</a></p>
 </body>
</html>
```

The page uses a Java bean called *PersonsBean*. I do not want to display the code here, but it creates a *List<PersonBean>*, which is initialized with all employees in the database. The most important thing here is to note how this list is used to dynamically create the table with a row for each employee. Here you should especially note the syntax for how to insert a loop with the element *c:forEach* on a JSP page.

There is yet another link on the start page where a member has the opportunity to change his password. I do not want to show this here as it does not add anything new. It is a form where the user must enter his employee number, password and new passwords, and the page uses a servlet to validate this data and possibly updates the database.

This web application is simple and are in more places simplified to a practical application, and of course, there could be more features such as search on employees, but it is a typical web application where users can edit data, which is stored in a database. In practice, many web applications are database applications, for example, thinking of a classic shopping cart.

However, before I completely leave the application there is a single standout. As mentioned, I uses a class *DB* to create a connection to the database. It requires at least a database user in the form of username and password. This information is often saved in a configuration file and I have used *web.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" … >
 <welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
 </welcome-file-list>
 <listener>
  <listener-class>changeaddress.ParamsFactory</listener-class>
 </listener>
 <context-param>
  <param-name>usr</param-name>
  <param-value>pa</param-value>
 </context-param>
 <context-param>
  <param-name>pwd</param-name>
  <param-value>Volmer_1234</param-value>
 </context-param>
</web-app>
```

It is generally considered to be a safe place as it is not sent to the client, and since the web container does not allow access to the file from the outside. However, this means that a Java class should be able to retrieve these parameters, and it can be done with a listener class that retrieves information when the application starts. Note that *web.xml* defines this listener as the *ParamsFactory* class, and the server will instantiate an object of this class that creates a *Properties* object with the context parameters as the key / value pair. This item is represented by the class

```
package changeaddress;

import java.util.*;

public abstract class WebParams
{
 private static Properties contextProperties = new Properties();

 public static String getParam(String key)
 {
  return contextProperties.getProperty(key);
 }
```

```
 public static void setProperties(Properties properties)
 {
  contextProperties = properties;
 }
}
```

there only have static members. The *Properties* object is created in *ParamsFactory* (listening Object):

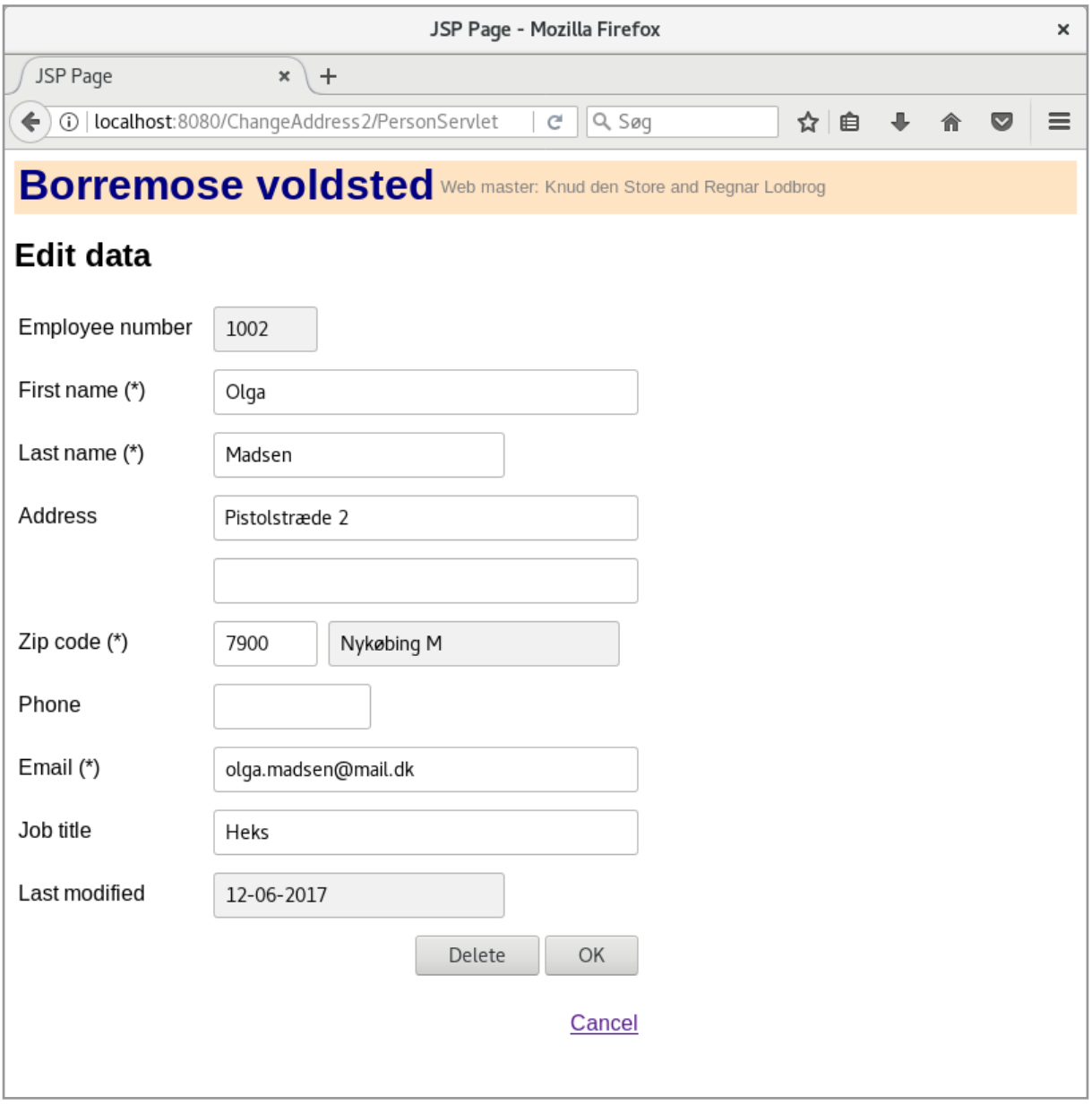
```
package changeaddress;

import java.util.*;
import javax.servlet.*;

public class ParamsFactory implements ServletContextListener
{
 public void contextDestroyed(ServletContextEvent event)
 {
 }

 public void contextInitialized(ServletContextEvent event)
 {
  Properties properties = new Properties();
  ServletContext servletContext = event.getServletContext();
  Enumeration<?> keys = servletContext.getInitParameterNames();
  while (keys.hasMoreElements())
  {
   String key = (String) keys.nextElement();
   String value = servletContext.getInitParameter(key);
   properties.setProperty(key, value);
  }
  WebParams.setProperties(properties);
 }
}
```

This method is automatically performed by the Glassfish server, and the following class can therefore connect to the database:

```
package changeaddress;

import java.sql.*;

public class DB
{
 public static Connection getConnection() throws SQLException
 {
  String usr = WebParams.getParam("usr");
  String pwd = WebParams.getParam("pwd");
  return DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/padata?useSSL=false", usr, pwd);
 }
}
```

EXERCISE 4

Start by creating a copy of the web application *ChangeAddress2*. You must expand the application to add a *Delete* button to the edit page:

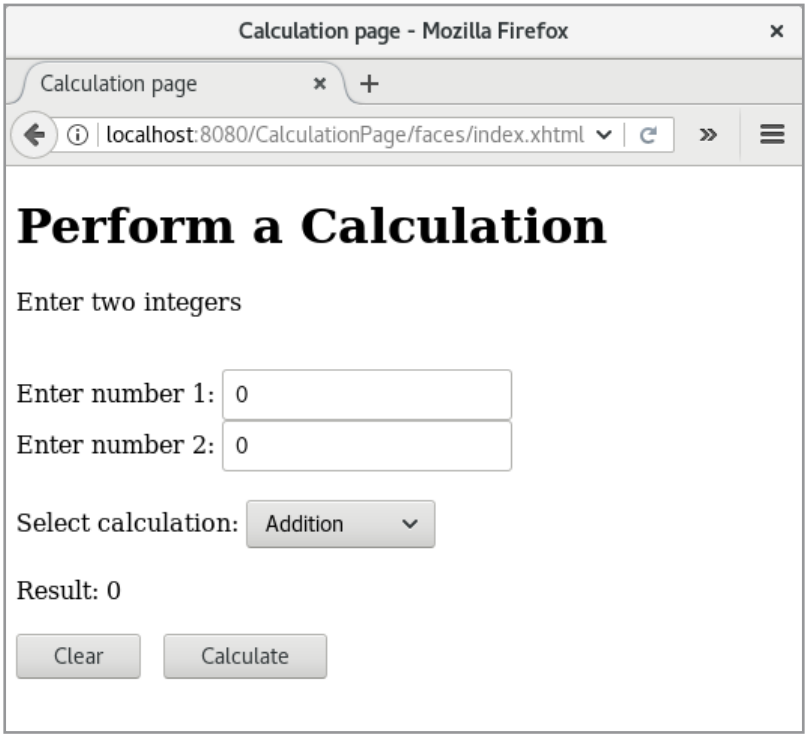


The button should only be there if you edit an existing employee. When you click the button, the employee must be deleted from the database. The easiest thing is to let the button post to a new servlet, which then deletes the employee.

6 JSF

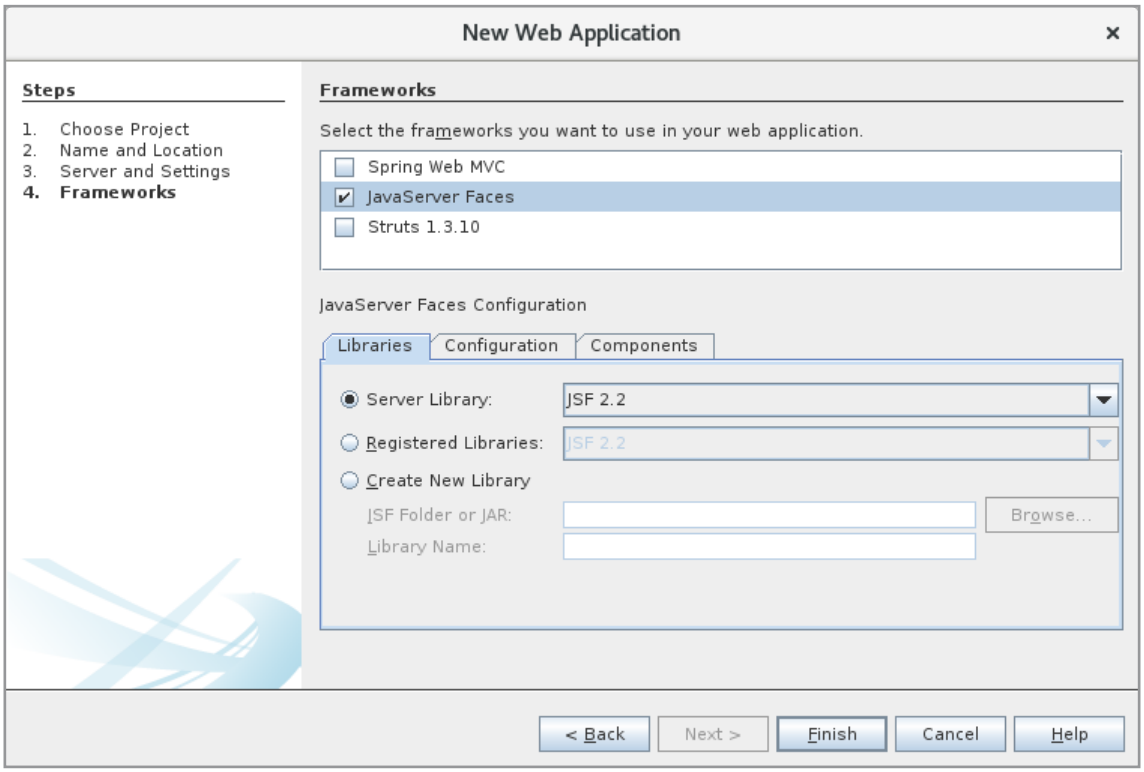
The whole idea behind JSP is to separate the presentation part from the application's business logic so that the individual pages of a web application are written as JSP pages, which then perform their data processing by sending requests to servlets, and as mentioned, the JSP page itself is translated into a servlet. The data to be manipulated by the JSP page is represented by a Java bean, which is a Java class whose properties can immediately be linked to the JSP page's fields. With this model you are able to develop effective dynamic websites. The technology has since been refined, and today wee talk about JSF, that stands for *Java Server Faces*, and the goal has been to make it easier to develop modern web applications and make it easier to use many new web technologies. A JSF page replaces the JSP pages, and a JSF page is linked to a special servlet, which is called a *FacesServlet*. One can also say that JSF is not much more than a natural development of a technology driven by the many new demands that constantly arise in the development of effective web applications.

I will start by showing how a web application with JSF can be written using NetBeans. The application is basically the same as I previously shown, where the user must enter two numbers, after which the program can perform a calculation:



The calculation is one of the four calculations types, and there are two buttons, one clear the fields (the two input fields and the result), while the other performs the calculation.

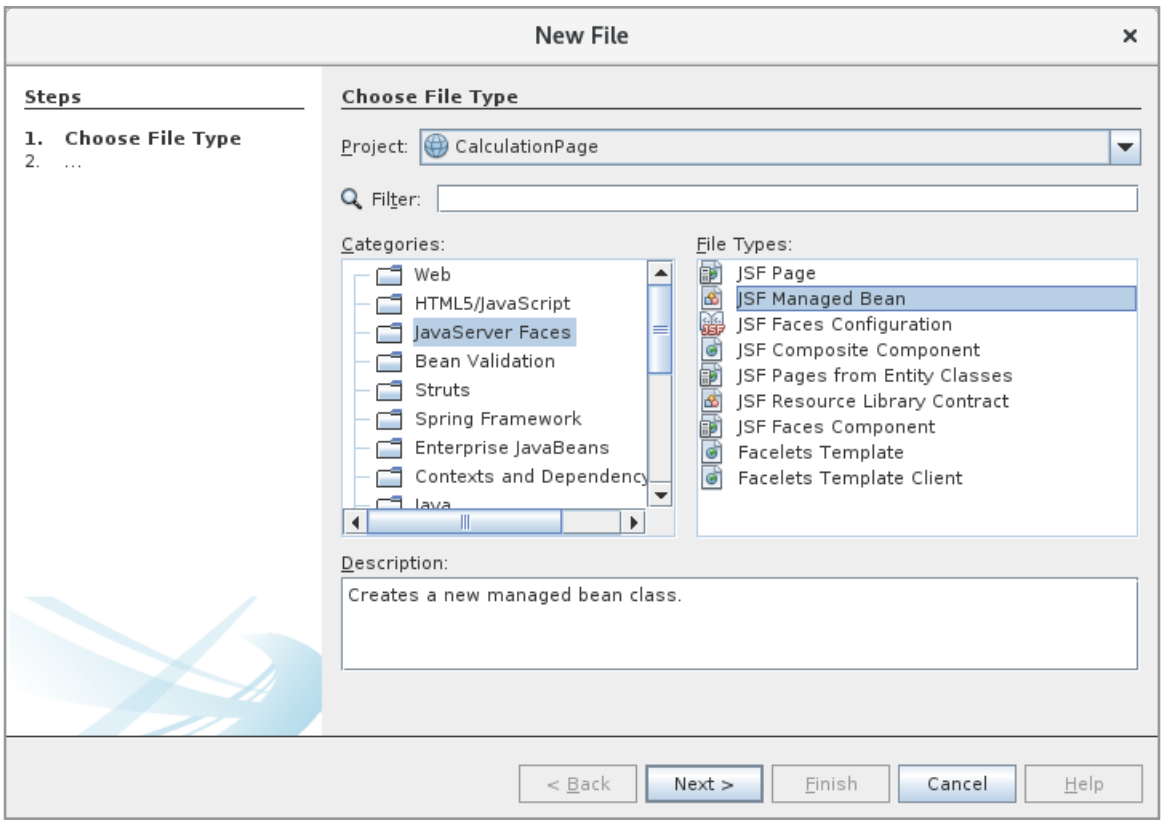
The project is called *CalculationPage* and is created in NetBeans as other web applications, but in the *Server Settings* window, I click *Next* and then I comes to the window for selecting *Frameworks*. Here you must tick the *JavaServer Faces* field (see below). The rest must remain as it is and when you click *Finish*, a JSF web application has been created that consists of a single file named *index.xhtml*:

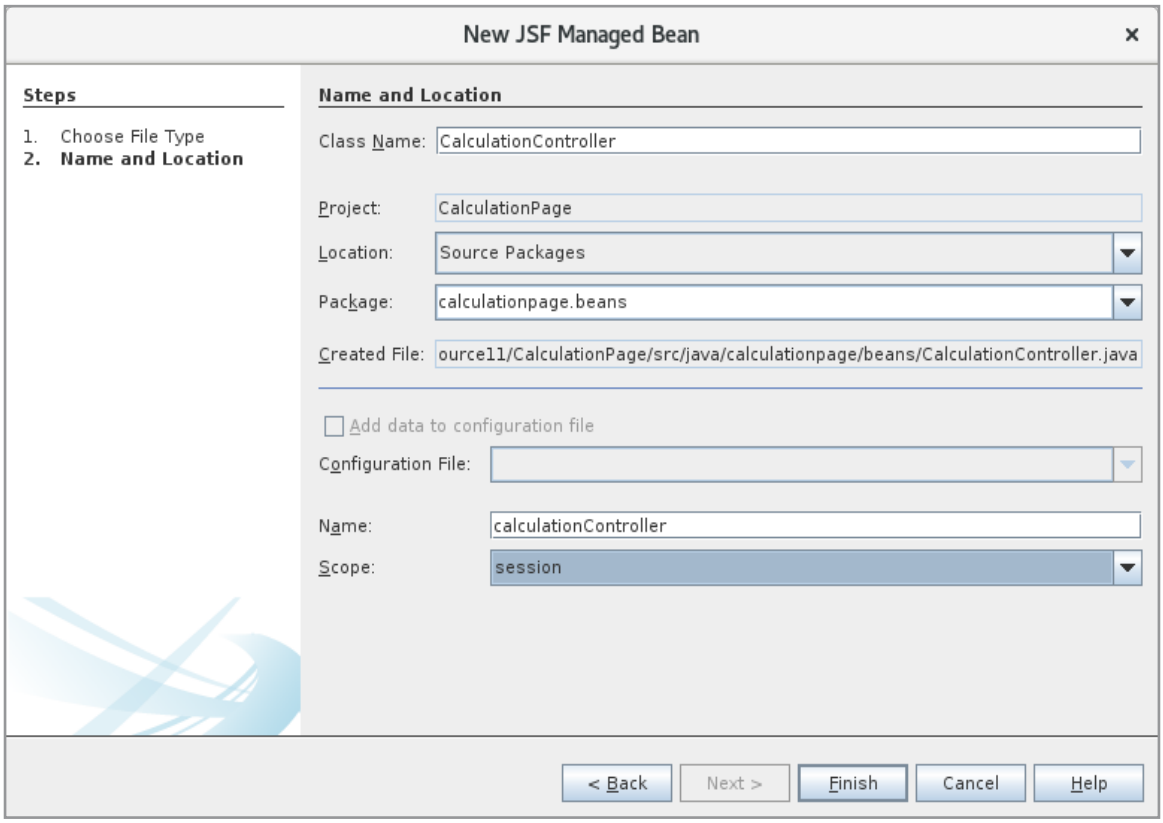



```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
  <title>Facelet Title</title>
 </h:head>
 <h:body>
  Hello from Facelets
 </h:body>
</html>
```

As you can see, it is an XML document with HTML syntax, and in fact it is a complete web application that can be translated and opened in the browser. The task is, of course, to modify this page so it displays a window as shown above.

Data to a JSP page is represented by a Java bean, and there will usually be a Java bean to any JSP page. The same goes for JSF pages, but instead, you're talking about a named bean (exactly a CDI named bean) or a controller, but basically, it's nothing but a usual Java bean with two annotations in front of the class. To associate such a named bean with the project, I have added a package *calculationpage.beans* as previously. Then I have right-clicked and selected *JSF Managed Bean* (see below). When I click OK, I get a window to create the desired class (see below). The name is *CalulationController* and you should note that the above package is selected. Also note the *Name* field, which is the name that the class is known in the JSF page. By default, NetBeans choose the class name starting with a lowercase letter, and there is seldom a reason to change it. Finally, note the field *Scope* where I have the choice session. When you click OK, NetBeans will create a named bean:




New JSF Managed Bean
Steps
1. Choose File Type
2. Name and Location
Name and Location
Class Name: CalculationController
Project: CalculationPage
Location: Source Packages
Package: calculationpage.beans
Created File: ource11/CalculationPage/src/java/calculationpage/beans/CalculationController.java
Add data to configuration file
Configuration File:
Name: calculationController
Scope: session
< Back
Next >
Finish
Cancel
Help

```
package calculationpage.beans;

import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;
@Named(value = "calculationController")
@SessionScoped
public class CalculationController implements Serializable
{
 public CalculationController()
 {
 }
}
```

Basically, it's nothing but a simple skeleton for a class, but you should notice that the class is serializable as other Java beans and of course the two annotations. Here you should note that the values are those that you have selected in the previous screen.

After you have attached a named bean to the project, the code must be written and it will be done in the same way as other beans. The result is the following where I have not shown the code for *get* and *set* methods (which are all trivial):

```
@Named(value = "calculationController")
@SessionScoped
public class CalculationController implements Serializable
{
 private static final String ADDITION = "Addition";
 private static final String SUBTRACTION = "Subtraction";
 private static final String MULTIPLICATION = "Multiplication";
 private static final String DIVISION = "Division";
 private long number1;
 private long number2;
 private long result;
 private String calculation;
 private List<SelectItem> calculations;

 public CalculationController()
 {
  clear();
  calculations = new ArrayList();
  calculations.add(new SelectItem(ADDITION));
  calculations.add(new SelectItem(SUBTRACTION));
  calculations.add(new SelectItem(MULTIPLICATION));
  calculations.add(new SelectItem(DIVISION));
 }
```

```
 public void clear()
 {
  number1 = 0;
  number2 = 0;
  result = 0;
  calculation = null;
 }

 public void calculate()
 {
  if (calculation.equals(ADDITION)) setResult(number1 + number2);
  else if (calculation.equals(SUBTRACTION)) setResult(number1 - number2);
  else if (calculation.equals(MULTIPLICATION)) setResult(number1 * number2);
  else if (calculation.equals(DIVISION))
   try
   {
    setResult(number1 / number2);
   }
   catch (Exception ex)
   {
    FacesMessage facesMsg = new FacesMessage(FacesMessage.SEVERITY_ERROR,
     "Invalid Calculation", "Invalid Calculation");
    FacesContext.getCurrentInstance().addMessage(null, facesMsg);
   }
 }
}
```

The class has four simple properties whose meaning should be self explanatory (the latter is used for the calculation operation to be performed), and there is also a list for objects to type *SelectionItem*, which represents items that the JSF page can display in a dropdown list. In addition to properties, the class has two methods, the first setting all properties to 0, while the last one performs a calculation. Since you can not divide by 0, the operation DIVISION can raise an exception. If it happens, a *FacesMessage* object will be created that represents an error message. I show below how it is used.

Then there's the JSF page, and here's the most to note:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://xmlns.jcp.org/jsf/core"
  xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
  <title>Calculation page</title>
```

```
</h:head>
<h:body>
 <f:view>
  <h1>Perform a Calculation</h1>
  <p>Enter two integers</p>
  <h:messages errorStyle="color: red" infoStyle="color: green"
   globalOnly="true"/>
  <br/>
  <h:form id="calulationForm">
   Enter number 1:
   <h:inputText id="number1" value="#{calculationController.number1}"/>
   <br/>
   Enter number 2:
   <h:inputText id="number2" value="#{calculationController.number2}"/>
   <br/>
   <br/>
   Select calculation:
   <h:selectOneMenu id="calculation"
    value="#{calculationController.calculation}">
    <f:selectItems value="#{calculationController.calculations}"/>
   </h:selectOneMenu>
   <br/>
   <br/>
   Result:
```

```
    <h:outputText id="result" value="#{calculationController.result}"/>
    <br/>
    <br/>
    <h:commandButton action="#{calculationController.clear()}"
     value="Clear"/>  
    <h:commandButton action="#{calculationController.calculate()}"
     value="Calculate"/>
   </h:form>
  </f:view>
 </h:body>
</html>
```

When you see the code, it is easy to understand, and immediately you notice that it is an XML document and that much of the code is similar to HTML. However, many new elements are used, which are not standard HTML elements, but elements that are defined in two tag libraries and are usually called JSF elements:

```
xmlns:f="http://xmlns.jcp.org/jsf/core"
xmlns:h="http://xmlns.jcp.org/jsf/html"
```

Such an element defines different attributes that can be initialized and the element is translated into standard HTML. For example, there is an element

```
<h:messages errorStyle="color: red" infoStyle="color: green" globalOnly="true"/>
```

which generally does not result in any code, but if there is an error in the calculation (division with 0) the following text is added to the document:

```
<ul><li style="color: red">Invalid Calculation </li></ul>
```

It is important to be aware that these new elements do not mean that you can not use standard HTML. This is possible in the same way as before. It's just a matter that the new elements giving new opportunities and opportunities to generate the desired HTML easier than to write it all from scratch. Note as an example how the dropdown list is defined:

```
<h:selectOneMenu id="calculation"
value="#{calculationController.calculation}">
 <f:selectItems value="#{calculationController.calculations}"/>
</h:selectOneMenu>
```

Note, in particular, how to refer to a bean property, and in principle it is done in the same way as in a JSP page, just use the character # instead of $. Looking at the above definition, it results in the following HTML:

```
<select id="calulationForm:calculation" name="calulationForm:calculation" size="1">
 <option value="Addition" selected="selected">Addition</option>
 <option value="Subtraction">Subtraction</option>
 <option value="Multiplication">Multiplication</option>
 <option value="Division">Division</option>
</select>
```

The page also defines a form element, but no action is defined. This action is instead associated with the definition of a button, such as:

```
<h:commandButton action="#{calculationController.clear()}" value="Clear"/>
```

Below is an example of the HTML that this JSF page generates:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
 <head id="j_idt2">
  <title>Calculation page</title>
 </head>
 <body>
  <h1>Perform a Calculation</h1>
  <p>Enter two integers</p>
  <br />
  <form id="calulationForm" name="calulationForm" method="post"
   action="/CalculationPage/faces/index.xhtml"
   enctype="application/x-www-form-urlencoded">
   <input type="hidden" name="calulationForm" value="calulationForm" />
   Enter number 1:
   <input id="calulationForm:number1" type="text" name="calulationForm:number1"
   value="0" />
   <br />
   Enter number 2:
   <input id="calulationForm:number2" type="text" name="calulationForm:number2"
   value="0" />
   <br />
   <br />
   Select calculation:
   <select id="calulationForm:calculation" name="calulationForm:calculation"
     size="1">
    <option value="Addition">Addition</option>
    <option value="Subtraction">Subtraction</option>
```

```
    <option value="Multiplication">Multiplication</option>
    <option value="Division">Division</option>
   </select>
   <br />
   <br />
   Result:
   <span id="calulationForm:result">0</span>
   <br />
   <br /><input type="submit" name="calulationForm:j_idt14" value="Clear" />
   <input type="submit" name="calulationForm:j_idt16" value="Calculate" />
   <input type="hidden" name="javax.faces.ViewState"
     id="j_id1:javax.faces.ViewState:0"
     value="-5991509489504870512:5541158679969395251" autocomplete="off" />
  </form>
 </body>
</html>
```

Most of it can be recognized. However, there are a few hidden fields whose value is difficult to interpret, but here you must remember how the HTML document is formed: The JSF page is being translated into a servlet, and this servlet sends the above HTML document as a response.

As a conclusion, a JSF application consists of a number of XHTML pages that contain primarily JSF elements as well as one or more named beans and optionally a configuration file called *faces-config.xml*. As the following examples will show, there may also be other Java classes, which typically represent model classes.

As part of the project, NetBeans has added a *web.xml* configuration file:

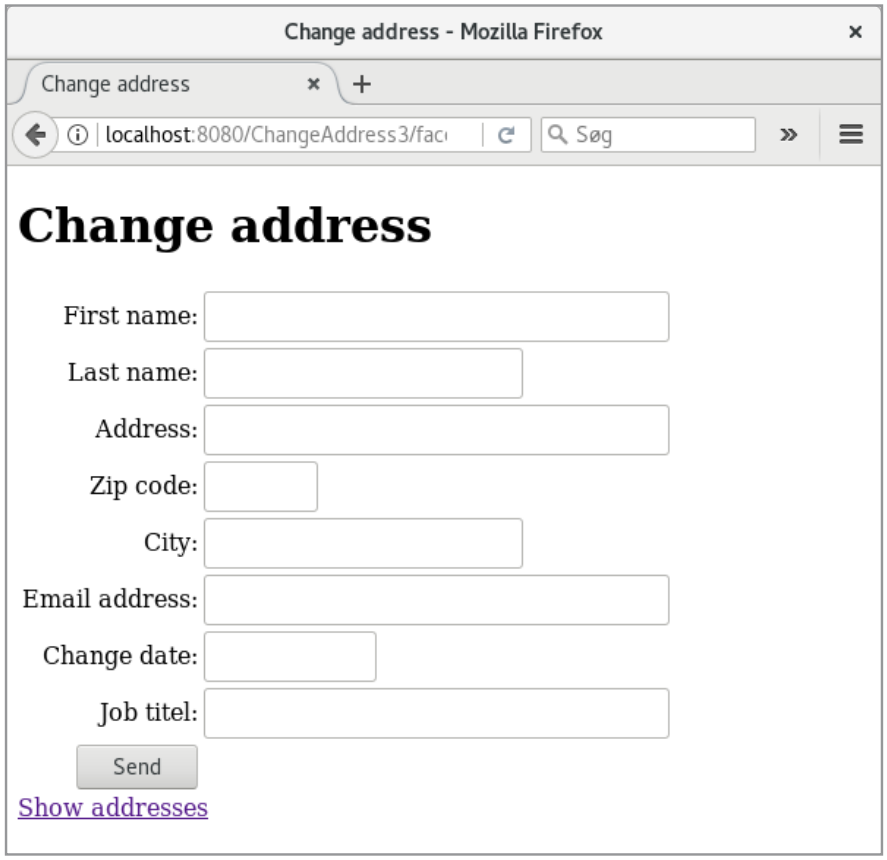
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
 http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
 <context-param>
  <param-name>javax.faces.PROJECT_STAGE</param-name>
  <param-value>Development</param-value>
 </context-param>
 <servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
 </servlet-mapping>
 <session-config>
  <session-timeout>
   30
  </session-timeout>
 </session-config>
 <welcome-file-list>
  <welcome-file>faces/index.xhtml</welcome-file>
 </welcome-file-list>
</web-app>
```

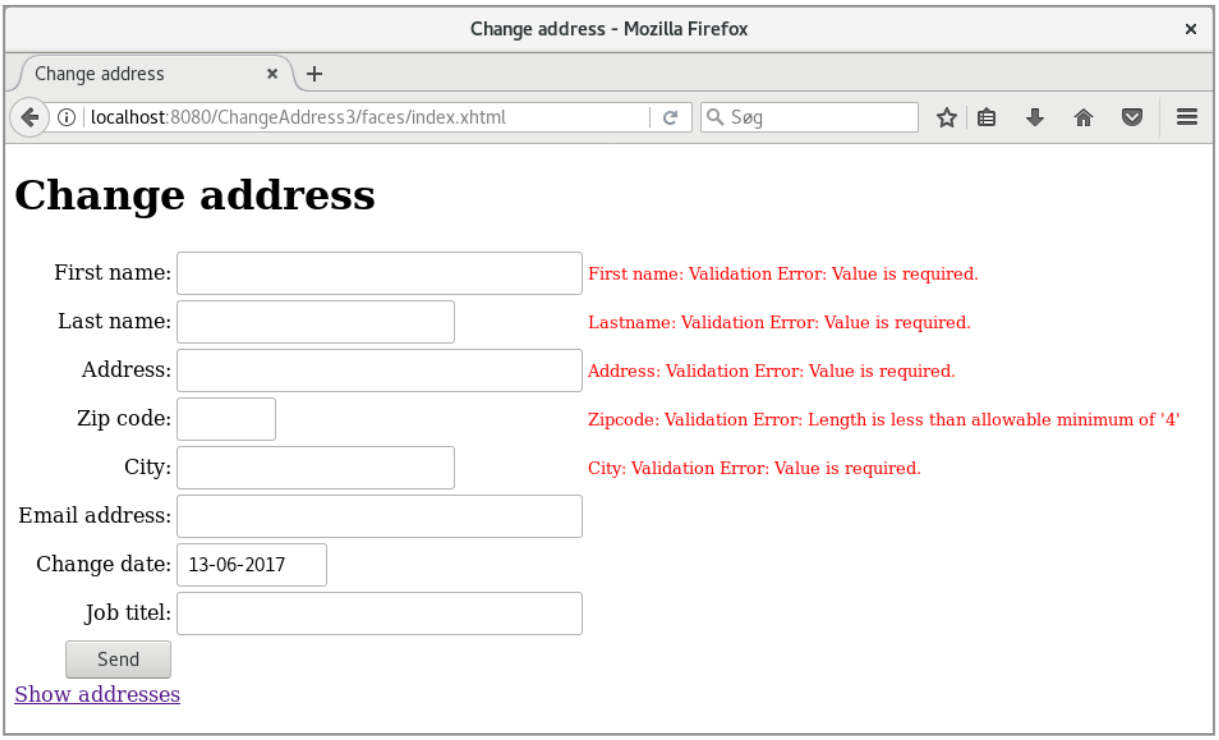
There is not much to note, and it is rarely maintained this file, but initially a context-param element has been inserted that defines a param *value* with the value *Development*, which means adding some debug information to the project. It is not interested in a finished application, and the value can then be changed to *Production*. Also note that *web.xml* defines the start page.

6.1 CHANGEADDRESS3

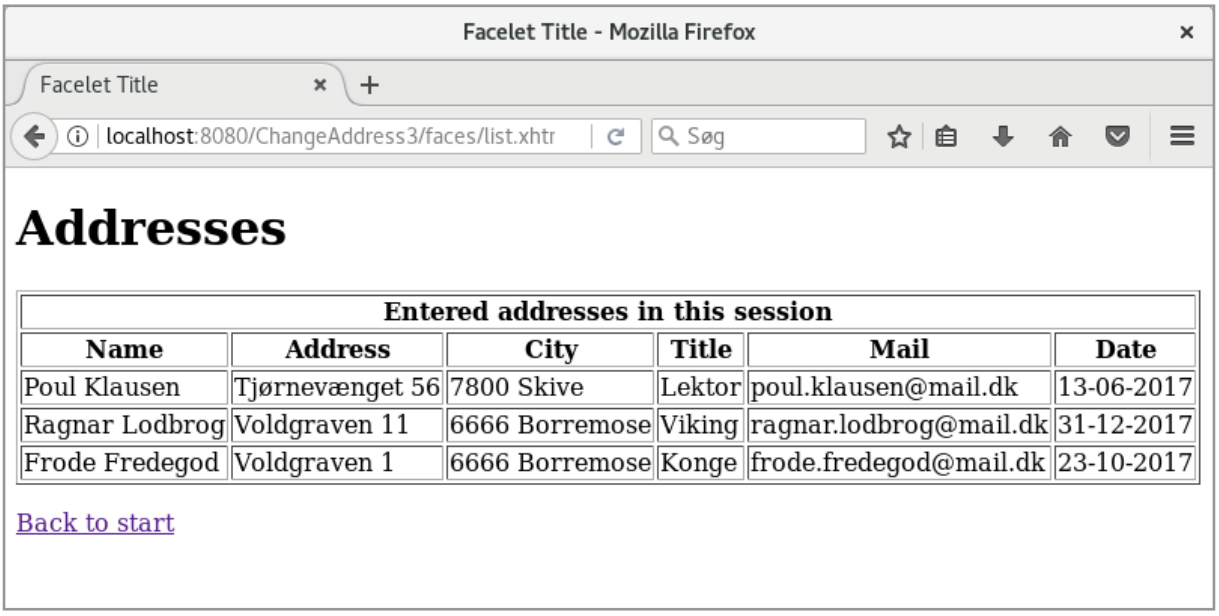
I want to show how to write a JSF application, similar to *ChangeAddress1*. I start with a new Web Application project, which I have called *ChangeAddress3*, and when I come to the window to select framework, I have chosen *JavaServer Faces* and selected session scope as above. The result is again a JSF page named *index.xhtml* which is the application's home page. If you open the application, you get the following window (see below).

If you enter a date and click on the *Send* button, the result is the window below. The reason is that the program validates the contents of the fields, and when a value has be entered for date, it is because this field is validated differently from the other fields. When there is no error message next to the Email address and Job title fields, it is because the first one must be empty while the last one is not validated at all.




Change address - Mozilla Firefox
Change address
localhost:8080/ChangeAddress3/faces/index.xhtml
Søg
Change address
First name:
First name: Validation Error: Value is required.
Last name:
Lastname: Validation Error: Value is required.
Address:
Address: Validation Error: Value is required.
Zip code:
Zipcode: Validation Error: Length is less than allowable minimum of '4'
City:
City: Validation Error: Value is required.
Email address:
Change date:
13-06-2017
Job titel:
Send
Show addresses

Once you have entered a legal address, it is saved in a list and the fields are blanked so that you can enter another address. If you click on the bottom link, there is a request for a page that shows an overview of the addresses that have been entered. The result could for example be:



Then to the code. Under *Source Packages*, three packages have been created:

1. *changeaddress.beans* which contains the application's beans, and there is only one that is a *named bean*
2. *changeaddress.models* which contains other Java class, and there is only one that represents an address
3. *changeaddress.validators* that contains classes to validate the user interface fields and there are two

I want to start with the class, which represents an address. The class is called *Person*, and is a usual model class and in fact it is a Java bean:

```
package changeaddress.models;
public class Person implements java.io.Serializable
{
 private String firstname;
 private String lastname;
 private String address;
 private String code;
 private String city;
 private String email;
 private String title;
 private String date;
```

```
 public Person()
 {
 }

 public Person(String firstname, String lastname, String address, String code,
  String city, String email, String title, String date)
 {
  …
 }

 public boolean isLegal()
 {
  return firstname != null && firstname.length() > 0 && lastname != null &&
   lastname.length() > 0 && date != null && date.length() > 0;
 }
}
```

There is not much to explain, and I have not shown the class's *get* and *set* methods, as they are all trivial. You must note the last method, which states that a *Person* object is legal if the object has a first name, a last name and a date. The purpose of the class is primarily to show that a web application can use model classes in the same way as other Java applications and here especially where the class is used from JSF pages.

As the next example, I will show a *Validator* class, which in this case is used to validate an input field and here the field's value must be a legal date:

```
package changeaddress.validators;

….

@FacesValidator(value ="dateValidator")
public class DateValidator implements Validator
{
 @Override
 public void validate(FacesContext facesContext, UIComponent uiComponent,
  Object value) throws ValidatorException
 {
  HtmlInputText htmlInputText = (HtmlInputText) uiComponent;
  String label = htmlInputText.getLabel() == null ||
   htmlInputText.getLabel().trim().equals("") ?
   htmlInputText.getId() : htmlInputText.getLabel();
  if (!isDate((String)value))
  {
   FacesMessage facesMessage =
```

```
    new FacesMessage(label + ": Illegal value for date");
   throw new ValidatorException(facesMessage);
  }
 }

 private boolean isDate(String text)
 {
  ….
 }
}
```

First, note that the class is decorated with an annotation telling that it is a *Validator* class. Also note that the class implements the *Validator* interface, which defines a single method *validate()*, which is the method that validates the field's value that is the parameter *value*. *uiComponent* is the component whose value must be validated and the method starts by determining the value of an attribute for this component (either its *ID* or *label*). Next, the value is tested using a private helper method called *isDate()*. I have not shown the code because it contains nothing but what I have shown in other books. If the value can not be validated correctly, a *FacesMessage* object is created and an exception is made with this object as a parameter. As a result, the error message appears in the window.

There is a corresponding *Validator* class, called *EmailValidator*. It is in principle identical to the above, and I do not want to show the code here.

Finally, the application has a named bean. The class is simple and consists primarily of *get* and *set* methods, where I have only shown a few examples:

```
package changeaddress.beans;

….

@Named(value = "indexController")
@SessionScoped
public class IndexController implements Serializable
{
 private Person person = new Person();
 private List<Person> persons = new ArrayList();

 public IndexController()
 {
 }

 public String getFirstname()
 {
  return person.getFirstname();
 }

 public void setFirstname(String firstname)
 {
  person.setFirstname(firstname);
 }

 ….

 public List<Person> getPersons()
 {
  return persons;
 }

 public void add()
 {
  if (person.isLegal())
  {
   persons.add(person);
   person = new Person();
  }
 }
}
```

Seen from *index.xhtml* is the most important the method *add()* as the method that is performed when clicking the *Send* button. It tests where a person is legal, and if that is the case, the method adds a *Person* object to the list, after which a new *Person* object is created.

I have chosen that this bean should have session scope and there are the following options:

1. *Request* which means that the bean only exists for the current HTTP request. That is, it must be created for each request.
2. *Session* which means that the bean object exists within a single user's HTTP session.
3. *Conversation* which means that the bean exists across multiple HTTP requests.
4. *Application* which means that the same bean is available to all users of the application across multiple user sessions.
5. *Dependent* which means that the bean is created every time there is a need.

Then *index.xhtml* which is a typical example of a form:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE … >
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
 <h:head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <title>Change address</title>
  <h:outputStylesheet library="css" name="styles.css"/>
 </h:head>
 <h:body>
  <h1>Change address</h1>
  <h:form>
   <h:panelGrid columns="3" columnClasses="rightalign,leftalign,leftalign">
    <h:outputLabel value="First name:" for="firstname"/>
    <h:inputText id="firstname" label="First name" required="true"
     style="width: 300px" value="#{indexController.firstname}" />
    <h:message for="firstname" class="error-message" />
    <h:outputLabel value="Last name:" for="lastname"/>
    <h:inputText id="lastname" label="Lastname" required="true"
     style="width: 200px" value="#{indexController.lastname}" />
    <h:message for="lastname" class="error-message" />
    <h:outputLabel value="Address:" for="address"/>
    <h:inputText id="address" label="Address" required="true"
     style="width: 300px" value="#{indexController.address}" />
    <h:message for="address" class="error-message" />
    <h:outputLabel value="Zip code:" for="code" />
    <h:inputText id="code" label="Zipcode"
```

```
style="width: 60px" required="false"
      value="#{indexController.code}">
      <f:validateLength minimum="4" maximum="4"/>
    </h:inputText>
    <h:message for="code" class="error-message" />
    <h:outputLabel value="City:" for="city"/>
    <h:inputText id="city" label="City" required="true" style="width: 200px"
      value="#{indexController.city}" />
    <h:message for="city" class="error-message" />
    <h:outputLabel value="Email address:" for="email"/>
    <h:inputText id="email" label="Email address" required="false"
      style="width: 300px" value="#{indexController.email}">
      <f:validator validatorId="emailValidator"/>
    </h:inputText>
    <h:message for="email" class="error-message" />
    <h:outputLabel value="Change date:" for="date"/>
    <h:inputText id="date" label="Change date" required="true"
      style="width: 100px" value="#{indexController.date}"
      immediate="true" onchange="submit()">
      <f:validator validatorId="dateValidator"/>
    </h:inputText>
    <h:message for="date" class="error-message" />
    <h:outputLabel value="Job titel: " for="title"/>
    <h:inputText id="title" required="false" style="width: 300px"
```

```
      value="#{indexController.title}" />
     <h:panelGroup/>
     <h:commandButton value="Send" action="#{indexController.add()}" />
    </h:panelGrid>
    <a href="list.xhtml">Show addresses</a>
  </h:form>
 </h:body>
</html>
```

When you see the code, it's mostly easy enough to understand, but there are some things you should notice. The application uses a stylesheet that is located in the directory

```
resources/css
```

You should note how to refer to this style sheet in the header. Otherwise, the form consists primarily of fields that are laid out in a *h:panelGrid*, which is translated into a HTML table with a row for each field and where there are three columns. The first column is the text, the second the input field, and the third a possible error message. The first row is for the first name and is defined as follows:

```
<h:outputLabel value="First name:" for="firstname"/>
<h:inputText id="firstname" label="First name" required="true"
  style="width: 300px" value="#{indexController.firstname}" />
<h:message for="firstname" class="error-message" />
```

The first statement defines the text, and at the same time it is stated that the text with the attribute *for* is attached to the input field *firstname*. When this field has a *label* attribute it is because it is used in connection with an error message. Also note how the value of the field is bound to a property in *indexController*. Finally, the last statement is for the error message, and you should note that the element with the *for* attribute is attached to the input field. If you consider the input field, it has an attribute

```
required="true"
```

which means that a validator is attached to the field, which simply means that a value for first name must be entered. If not, the message field displays an error message. It is an example of a predefined *Validator*, and there are some of them. For example, if you look at the zipcode field, it also has a *Validator* that validates the value of the field to have a length within an interval. It is actually a very useful validator. If you consider the input field to date, it uses a custom *Validator*, which is a *DateValidator* object:

```
<h:inputText id="date" label="Change date"
required="true" style="width: 100px"
 value="#{indexController.date}" immediate="true" onchange="submit()">
 <f:validator validatorId="dateValidator"/>
</h:inputText>
```

In general, validation is performed when you click on the submit button – and before the action method is performed. It is important to be aware that validation takes place on the server side and that a request is made to the server. If all fields can not be validated, the action method is not executed, but there is a return to *index.xhmtl* and an error messages are displayed. You should note that in this case, the input field has two new attributes:

```
immediate="true" onchange="submit()"
```

They means that the content of the field must be validated immediately after the field loses focus, and thus not only when submitting the form. This option is not used as often as a request to the server still occurs and because you can do it better with ajax (see the next book). Lastly note the input field for title and that it has no validator.

As mentioned, the components in this example are placed using a *panelGrid*, which is a JSF element that is translated into an HTML table. There is also a *panelGroup* element used to collect multiple items as a single cell in the table. Here is an empty *panelGroup*, which simply means that an empty cell occurs.

Clicking on the bottom link will as mentioned above display a page with an overview of the addresses that were entered. It's also a JSF page (and thus a FaceLet) called *list.xhmtl*:

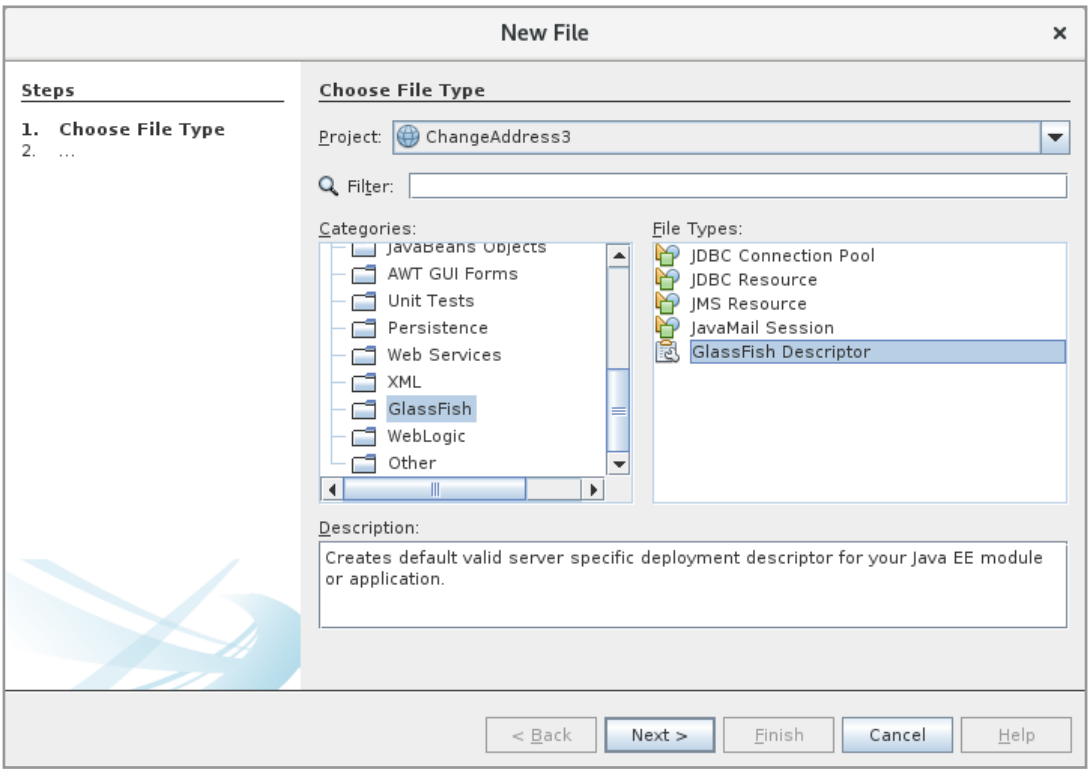
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml" … >
 <h:head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <title>Facelet Title</title>
  <h:outputStylesheet library="css" name="styles.css"/>
 </h:head>
 <h:body>
  <h:form>
   <h1>Addresses</h1>
   <h:dataTable id="addrTable" var="addr" border="1"
    value="#{indexController.persons}"
    rendered="#{indexController.persons.size() > 0}">
    <f:facet name="header">
     Entered addresses in this session
```

```
</f:facet>
<h:column id="nameCol">
 <f:facet name="header">Name</f:facet>
 <h:outputText id="name" value="#{addr.firstname} #{addr.lastname}"/>
</h:column>
<h:column id="addrCol">
 <f:facet name="header">Address</f:facet>
 <h:outputText id="addr" value="#{addr.address}"/>
</h:column>
<h:column id="cityCol">
 <f:facet name="header">City</f:facet>
 <h:outputText id="city" value="#{addr.code} #{addr.city}"/>
</h:column>
<h:column id="jobCol">
 <f:facet name="header">Title</f:facet>
 <h:outputText id="job" value="#{addr.title}"/>
</h:column>
<h:column id="mailCol">
 <f:facet name="header">Mail</f:facet>
 <h:outputText id="mail" value="#{addr.email}"/>
</h:column>
<h:column id="dateCol">
 <f:facet name="header">Date</f:facet>
 <h:outputText id="date" value="#{addr.date}"/>
```

```
    </h:column>
   </h:dataTable>
   <p><a href="index.xhtml">Back to start</a></p>
  </h:form>
 </h:body>
</html>
```

The code is an example that dynamically builds a table. You should note the syntax and that the page uses the same named bean (same controller) as *index.xhmtl*. In particular, you should notice how to iterate all person objects in the bean object's list.

Then the application is complete, but there is a single outstanding. The application may display national characters incorrectly. The problem is, that Glassfish may use an incorrect default encoding. If you right-click on WEB-INF and choose new, you can add a descriptor to the server:



The file is called *glassfish-web.xml* and the content must be:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-web-app PUBLIC
 "-//GlassFish.org//DTD GlassFish Application Server 3.1 Servlet 3.0//EN"
 "http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
```

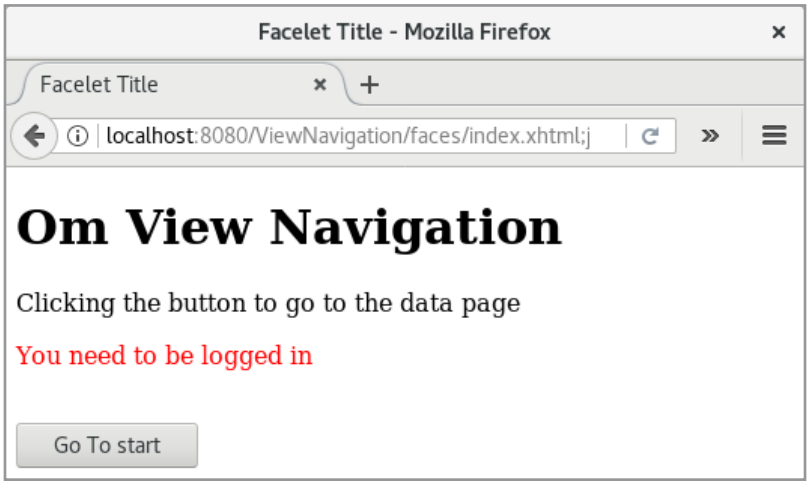
```
<glassfish-web-app error-url="">
 <class-loader delegate="true"/>
 <jsp-config>
  <property name="keepgenerated" value="true">
   <description> … </description>
  </property>
 </jsp-config>
 <parameter-encoding default-charset="UTF-8" />
</glassfish-web-app>
```

where I have added the blue line. Then the application is the finish.

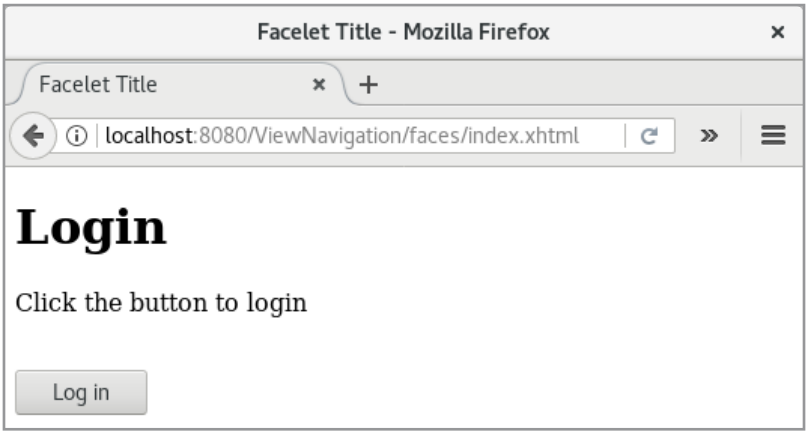
6.2 PAGE NAVIGATION

A typical web appilation consists of many pages (or views) and works while the user navigating between different pages often by performing an action with either a *h:commandButton* or *h:commandLink* element. In both cases, it means calling a named bean. For large applications, it can actually be complicated to control, especially because navigating a page may depend on a condition. For these reasons, a special descriptor file has been introduced that specifies how navigation can take place. The example *ViewNavigation* has the following *index.xhtml*:

where there are three buttons, each of which has the type *h:commandButton* and has an action. If you click on the first button, you get the following window, which is another view (called *info.xhtml*):



and clicking here on the button, you return to *index.xhtml*. If you click on the middle button on the start screen, you get the following window:



called *login.xhtml* and clicking the button you comes to the *data.xhtml* window:

Clicking this button will return to the start page (*index.xhtml*). If you click on the middle button again, you'll get directly to *data.xhtml* without having to go to *login.xhtml*. The last button on the start page is used for logout, and if you click on it and then on the middle button, you again has to go to *login.xhtml*.

To control it, a named bean, called *NavigationController*, is used:

```
package viewnavigation.beans;

import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;
```

```
@Named(value = "navigationController")
@SessionScoped
public class NavigationController implements Serializable
{
 private boolean loggedIn = false;

 public NavigationController()
 {
 }

 public String toStart()
 {
  return "START";
 }

 public String toInfo()
 {
  return "INFO";
 }

 public String nextPage()
 {
  if (loggedIn) return "data"; else return "login";
 }

 public String login()
 {
  loggedIn = true;
  return "LOGIN";
 }

 public void logout()
 {
  loggedIn = false;
 }
}
```

It defines methods that return a string and can be called as an action. The class has a variable to simulate if the user is logged in. There are two types of navigation. The first is called *explicit navigation*, where the action method references a string (a key) that refers to a *faces-config.xml* configuration file, which then indicates which view should be displayed. The other is called *implicit navigation*, where the action method returns the name of the current view (but without extension). For example, if you consider the *toStart()* and *toInfo()* methods, they return a string used in *faces-config.xml* and are examples of explicit navigation. As another example, the method *nextPage()* is an example of implicit navigation as it simply returns the name of a view (which depends on a condition). Also note the two last methods, the first one also being used as an action method for explicit navigation, but it first modifies the loggedIn variable to simulate that the user is logged in. The last one is actually an action method, but does not return anything, which means that there is no forward to another view.

Then there is the configuration file (added to *WEB-INF*):

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.2" … >
 <navigation-rule>
  <from-view-id>/index.xhtml</from-view-id>
```

```
  <navigation-case>
   <from-outcome>INFO</from-outcome>
   <to-view-id>/info.xhtml</to-view-id>
  </navigation-case>
 </navigation-rule>
 <navigation-rule>
  <from-view-id>/info.xhtml</from-view-id>
  <navigation-case>
   <from-outcome>START</from-outcome>
   <to-view-id>/index.xhtml</to-view-id>
  </navigation-case>
 </navigation-rule>
 <navigation-rule>
  <from-view-id>/login.xhtml</from-view-id>
  <navigation-case>
   <from-outcome>LOGIN</from-outcome>
   <to-view-id>/data.xhtml</to-view-id>
  </navigation-case>
 </navigation-rule>
</faces-config>
```

In this case, three navigation rules are defined. The first is interpreted as follows. If you come from (the action has taken place) *index.xhtml*, there is a single navigation case that says that if the value (key) is *INFO*, you must navigate to the *info.xhtml* page. You should note that a navigation rule may have more (many) navigation-case entries.

By using a *faces-config.xml* file, you can completely control how to navigate between individual views in a web application, but it can mean a very large file where you need to write a lot. Therefore, the concept of implicit navigation has been introduced, which does not use the configuration file.

Below is the code of *index.xhtml*:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
  <title>Facelet Title</title>
 </h:head>
 <h:body>
  <h:form id="form">
   <h1>Hello World</h1>
   <h2>Demonstrates page navigation</h2>
```

```
    <br/>
    <h:commandButton style="width: 150px" action="#{navigationController.toInfo}"
     value="Go To Information"/> 
    <h:commandButton style="width: 150px"
     action="#{navigationController.nextPage}" value="Goto data page"/> 
    <h:commandButton style="width: 150px" action="#{navigationController.logout}"
     value="Logout"/>
  </h:form>
 </h:body>
</html>
```

There is not much to explain, but you should note that the view has three commands. The first one will always forwards to *info.xhtml* (by explicit navigation). The other will forwards to either *login.xhtml* or *data.xhtml* with the help of implicit navigation. Finally, the last one will not make a forward as the method *logout()* does not return a value. Finally, I want to show *data.xhtml*:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
  <title>Facelet Title</title>
 </h:head>
 <h:body>
  <h:form>
   <h1>The data</h1>
   <ul>
    <li>Gorm den Gamle</li>
    <li>Harald Blåtand</li>
    <li>Svend Tverskæg</li>
   </ul>
   <br/>
   <h:commandButton action="index" value="Go to start"/>
  </h:form>
 </h:body>
</html>
```

Here you should note that implicit navigation is used, but instead of calling a method, the name of the view is directly listed as an argument for the action attribute.

PROBLEM 2

In this task, you must write a web application for a simple guestbook, like exercise 2. The program should fundamentally open the same form, but unlike exercise 2, data must be stored in a database. Start by creating a database called *guestbook*. The database should only have a single table that can be created with the following script:

```
use guestbook;
drop table if exists guests;

create table guests
(
 id int auto_increment primary key,
 name varchar(100) not null,
 addr varchar(100) not null,
 code char(4) not null,
 mail varchar(100),
 text text not null,
 date date
);
```

where the date is automatically assigned when writing in the guestbook.

It is a requirement that the program must be written using *JavaServer Faces*. In addition to the form to write in the guestbook, there must be a JSF page to which the home page links that shows the content of the guestbook.

6.3 TEMPLATES

As mentioned, a web application will consist in practice of many pages or views, and typically they are similar to each other and have a common design. In addition, templates can be used, and the following application shows how. Specifically, I will show the development of an application step by step, including how to use templates. The example also shows a number of other details regarding web application development, and especially the ability to use composite components, which are code that can be used in multiple views. In addition, the example shows how to apply images. The application is called *PaWeb* and will illustrate a very simple personal website where a person (and here the author) wishes to share private information using a public website.

The application is created as all other web applications in this chapter, and a *FaceLet* is created, named *index.xhtml*, which serves as the applications launcher. Note that after the application has been created, in addition to *index.xhtml*, a descriptor *web.xml* has also been created, which basically states that it is a JSF application. I have edited *index.xhtml*, so the finished code is the following, which just shows a welcome page:

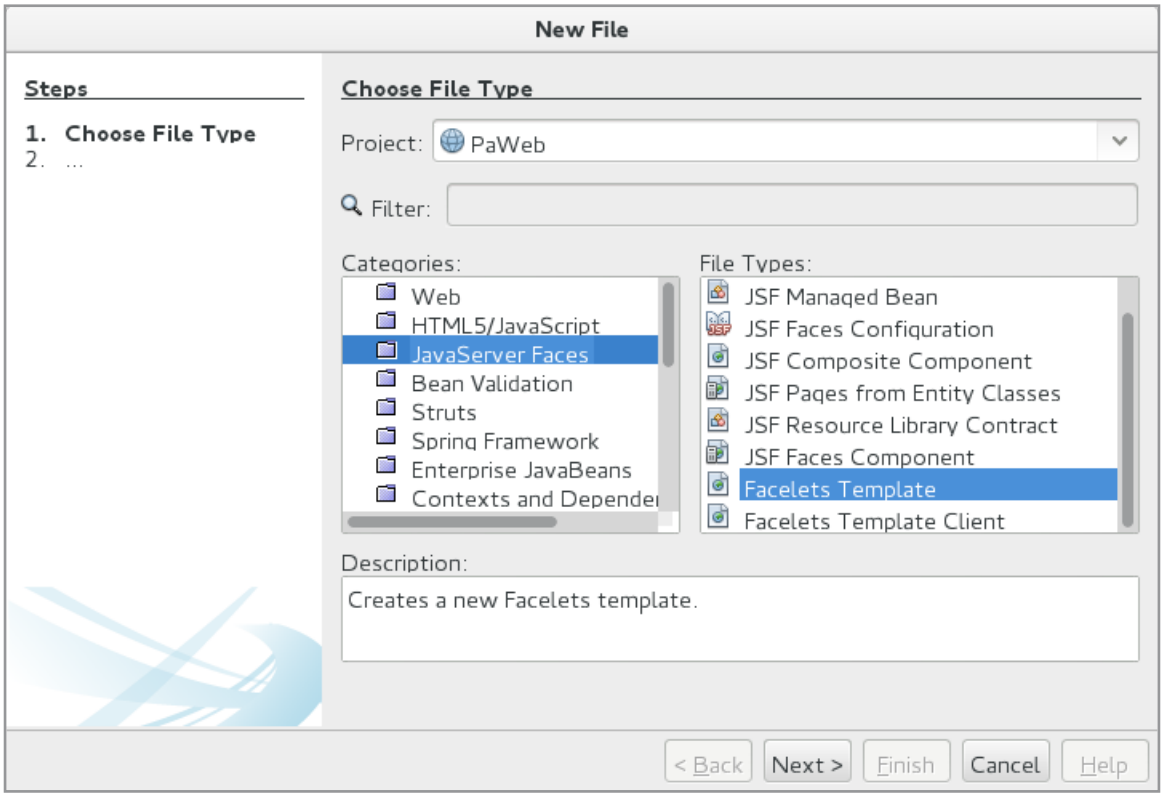
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//
EN" … > <html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
  <title>Facelet Title</title>
 </h:head>
 <h:body style="color: darkblue; background: bisque">
  <h:form>
   <div style="font-size: 144pt; font-weight: bold; text-align: center">
    Welcome</div>
   <div style="font-size: 72pt; text-align: center">to my website</div>
   <div style="font-size: 36pt; text-align: center">
    The site about big and small in my everyday life</div>
   <p style="text-align: center"><h:commandLink
    value="It sounds interesting so read on here" action="gallery"/></p>
   <p style="text-align: center">Poul Klausen</p>
  </h:form>
 </h:body>
</html>
```

There is not much new to explain, but you should note that the code is a mix of common HTML elements and JSF elements of the form

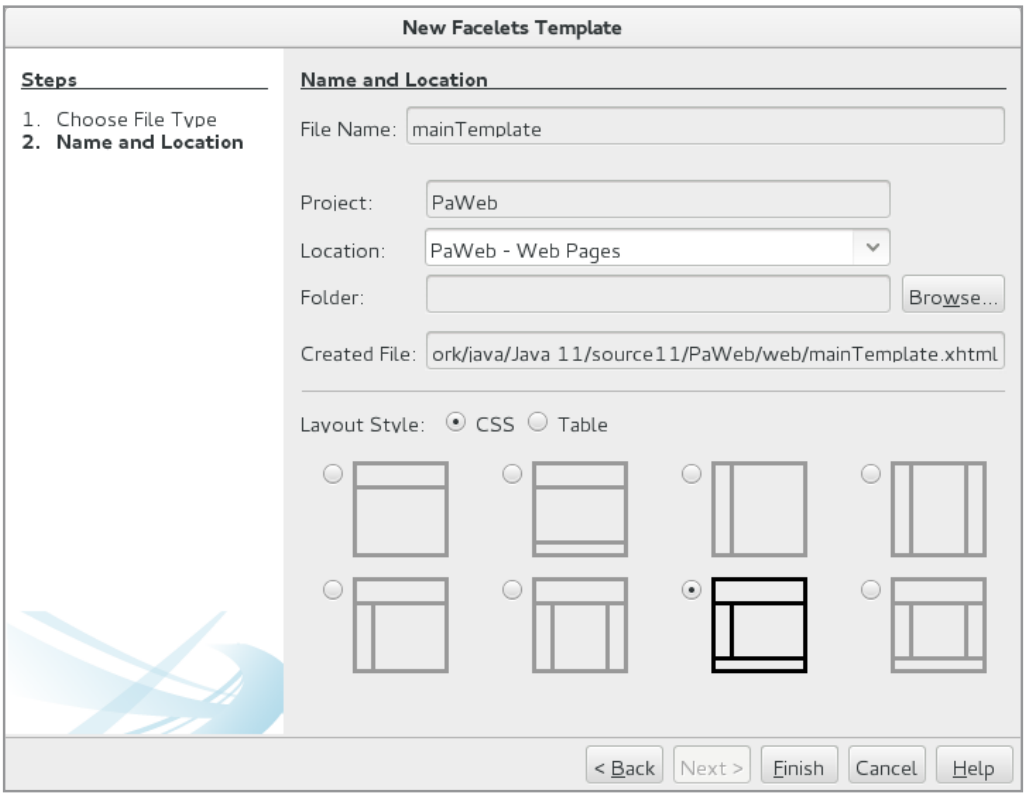
<h:code … >….</h:code>

JSF elements are server code and thus elements translated by the server to HTML and send as part of server's response. There are many such JSF elements, and there is nothing wrong with mixing JSF elements and HTML elements. Also note that the above JSF view uses styles. Styles are dealt with in the next book, but when you see the above code, it is easy enough to interpret what each style means. Note that using styles you can define a very large font. The view is a form since it contains a single form element, which has a *commandLink* element that is translated by the server into a common link element. The item has an action, which is the view that should be displayed if you click the link and it is called *gallery.xhtml*. Although this view is not created, the application can easily be translated and opened in the browser.

I will then show how to define a template that defines an overall design for a view. In NetBeans, right-click on Web Pages and choose *New* and *Facelets Template*:



When you click OK, you will get to the following window:



where you must choose a name and partly choose the design. I have entered the name *mainTemplate* and selected a design that divides the view into four sections (top, left, bottom and center). When you click *Finish*, NetBeans creates a JSF page:

`mainTemplate.xhtml`

In addition, a directory is created

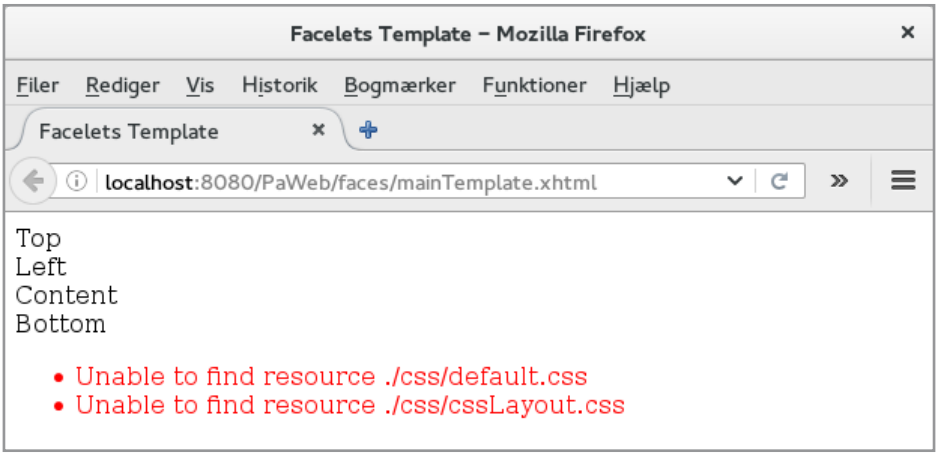
`resources/css`

and including two style sheets. As the name says, *resources* are a directory for resources, such as style sheets, but it can also be script codes or images. If you examine *mainTemplate.xhtml*, the code is as follows:

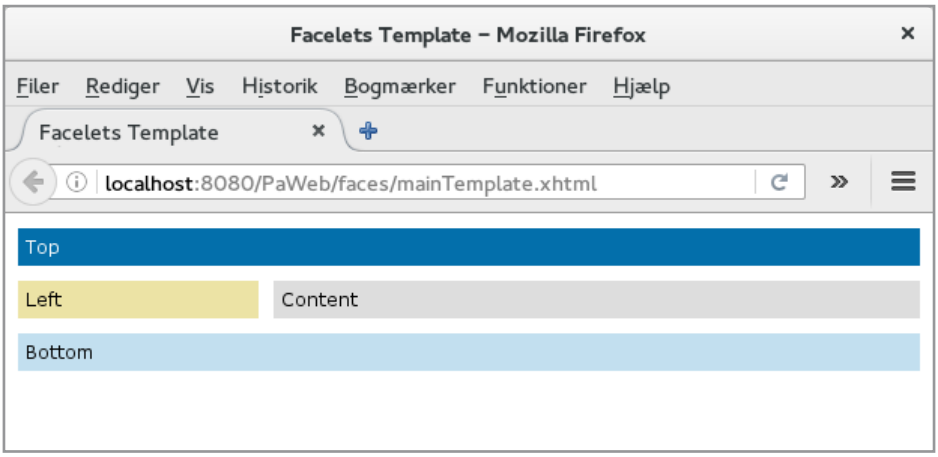
```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//
EN" … > <html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <h:outputStylesheet name="./css/default.css"/>
  <h:outputStylesheet name="./css/cssLayout.css"/>
  <title>Facelets Template</title>
 </h:head>
 <h:body>
  <div id="top">
   <ui:insert name="top">Top</ui:insert>
  </div>
  <div>
   <div id="left">
    <ui:insert name="left">Left</ui:insert>
   </div>
   <div id="content" class="left_content">
    <ui:insert name="content">Content</ui:insert>
   </div>
  </div>
  <div id="bottom">
   <ui:insert name="bottom">Bottom</ui:insert>
  </div>
 </h:body>
</html>
```

You must note a section is defined as an *ui:insert* element where, as default, a text is inserted and it is this text that the FaceLets using the template will replace with their own content. In fact, it's a common JSF view that can be opened in the browser, and if you do, it is the result as shown below. It does not matter so much, and a template is not meant to be a true view, but as a design that other JSF views can apply. As you can see, the view gives two error messages, and it is because the two lines in the header referring to the two style sheets need to be changed:

```
<h:head>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
 <h:outputStylesheet library="css" name="default.css"/>
 <h:outputStylesheet library="css" name="cssLayout.css"/>
 <title>Facelets Template</title>
</h:head>
```



If you then open the page again, the result is:



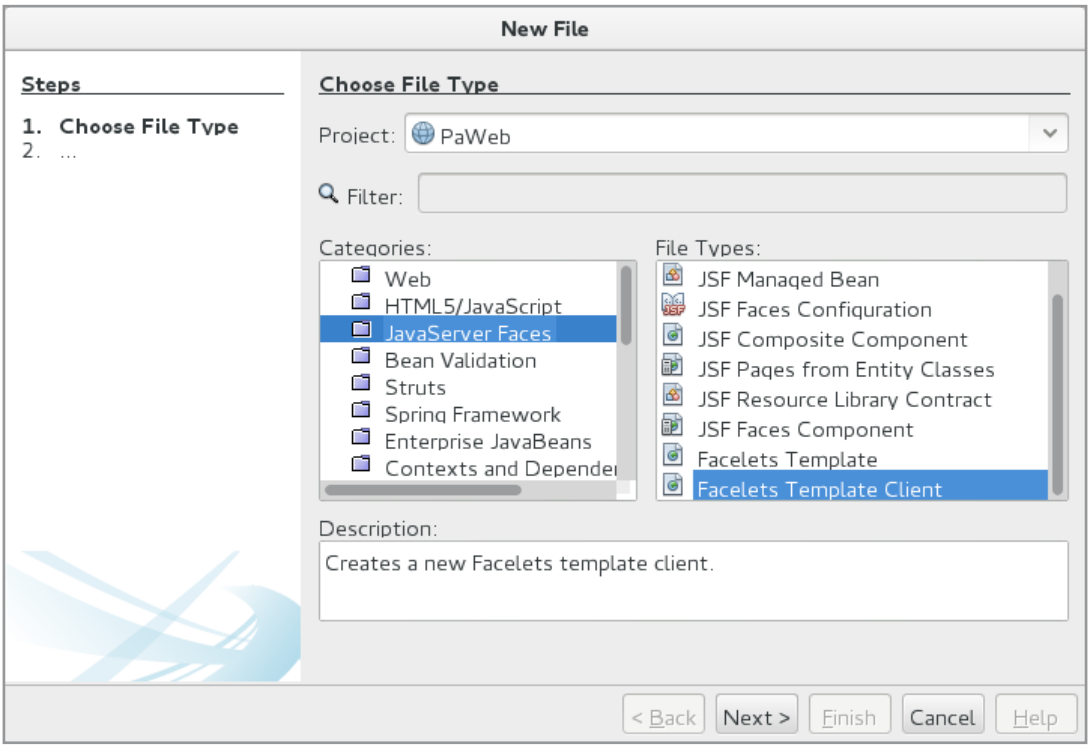
and that is because the template now uses the two style sheets, and they can subsequently be used by all views that use that template. I do not want to show the two style sheets here, but you are encouraged to examine the contents of the two files, and if you compare with the above window, you can easily interpret the content.

As can be seen from the above, a template (in this case) divides the view into four sections, which are defined by an element of the form:

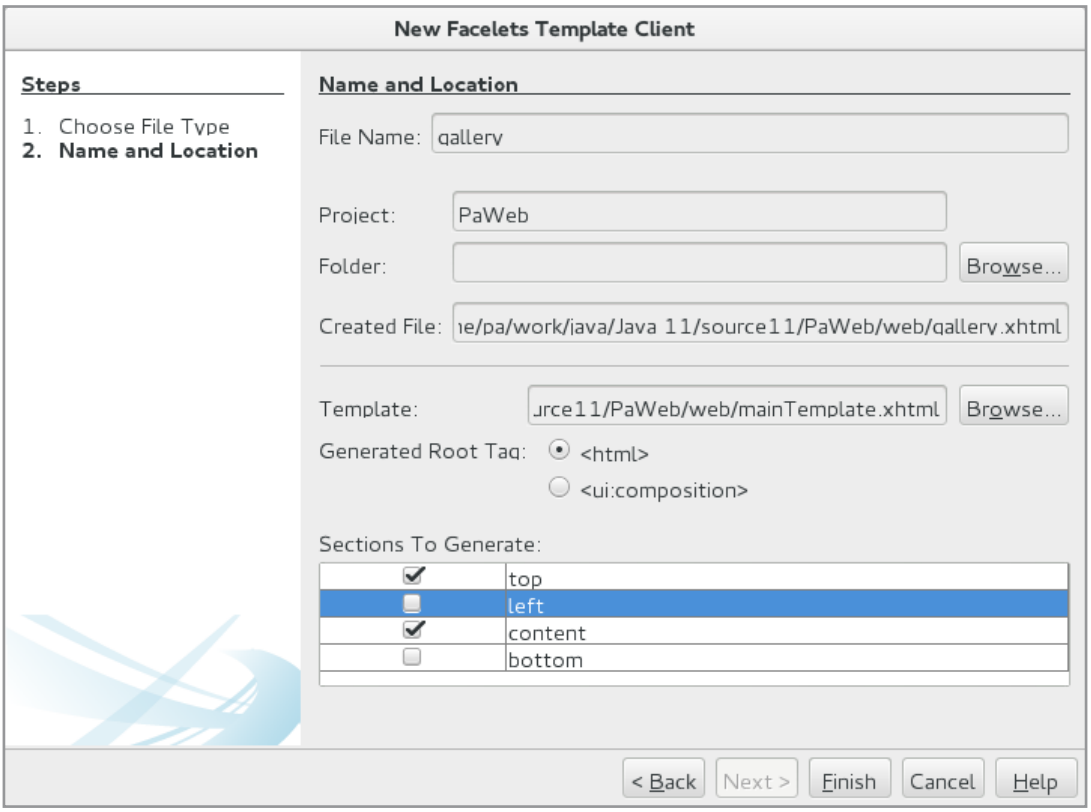
```
<ui:insert name="top">Top</ui:insert>
```

These items can be modified to have a more meaningful content, but other views are supposed to include code in one or more of the four areas.

I will now add another view to the application called *gallery.xhtml*, but it should be created as a *Facelet Template Client*:



When you click *Next*, you get the following window where you enter the name and select the template to be used:



and furthermore, you can select which sections in the template as this view are to insert code in. The result is the following view:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
 <body>
  <ui:composition template="./mainTemplate.xhtml">
   <ui:define name="top">
    top
   </ui:define>
   <ui:define name="content">
    content
   </ui:define>
  </ui:composition>
 </body>
</html>
```

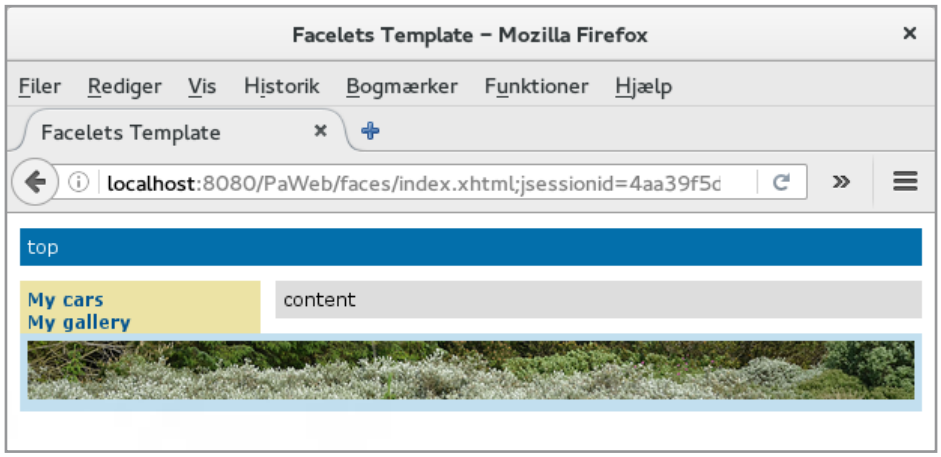
You should note that it is a JSF page and thus a view, but it uses the template and inserts code in the top and the center. If you run the application, open the start page (*index.xhtml*) and click on the link, opens the view *gallery.xhtml*, which displays the same window as shown above.

The task now is to get something more meaningful in the four areas. I will state with the template. First, I have created a sub directory *images* under *resources* and for this I have copied a picture called *thy.jpg*. Then here I changed *mainTemplate.xhtml* to the following:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//
EN" … > <html xmlns="http://www.w3.org/1999/xhtml" … >
 <h:head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <h:outputStylesheet library="css" name="default.css"/>
  <h:outputStylesheet library="css" name="cssLayout.css"/>
  <title>Facelets Template</title>
 </h:head>
 <h:body>
  <div id="top">
   <ui:insert name="top">
    <h1>Poul's Website</h1>
   </ui:insert>
  </div>
  <div>
```

```
    <div id="left">
     <ui:insert name="left">
      <h:form>
       <h:commandLink value="My cars" action="mycars" /><br/>
       <h:commandLink value="My gallery" action="gallery" /><br/>
      </h:form>
     </ui:insert>
    </div>
    <div id="content" class="left_content">
     <ui:insert name="content">Content</ui:insert>
    </div>
   </div>
   <div id="bottom">
    <ui:insert name="bottom">
     <h:graphicImage library="images" style="width: 100%" name="thy.jpg"/>
    </ui:insert>
   </div>
  </h:body>
</html>
```

That is, I have inserted code into three of the four areas. If you open the application and click on the link on the home page, you get the following results (which is *gallery.xhtml*):



In the left area there is a form that contains a menu with *commandLink* elements. The first links to the view itself, while the other links to a view that has not yet been created. You should note the bottom and how to insert an image. Note, in particular, how to style the image to fill the entire window. Finally, there is the top that contains just *h1* element, but apparently this item is not used. The reason is that the top is overridden by *gallery.xhtmp*, which inserts a default content in the top section, thus overriding the result from the template.

Next task is to make the view *galley.xhtml* complete. The content must be a list of links to images and you can see a picture by clicking a link. The view, like other views, has a controller, but first I want to define a class that can represent an image:

```
package paweb.models;

public class Data
{
 private String name;
 private String text;

 public Data(String name, String text)
 {
  this.name = name;
  this.text = text;
 }

 public String getName()
 {
  return name;
 }
```

```
 public String getText()
 {
  return text;
 }
}
```

where the class is located in the package *paweb.models*. It's a fairly simple model class and a usual Java class. *Text* is the value to be displayed on the screen, while *name* is the name of the picture (or something else) that the class should represent. In practice, there would probably be more properties attached to an image, and to illustrate it, I have defined following derived class;

```
package paweb.models;

public class Photo extends Data
{
 public Photo(String name, String text)
 {
  super(name, text);
 }

 public String toString()
 {
  return getText();
 }
}
```

Then I have written a controller class to the gallery view:

```
package paweb.beans;

import java.util.*;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;

import paweb.models.*;

@Named(value = "galleryController")
@SessionScoped
public class GalleryController implements Serializable
{
 private List<Photo> photos = new ArrayList();
```

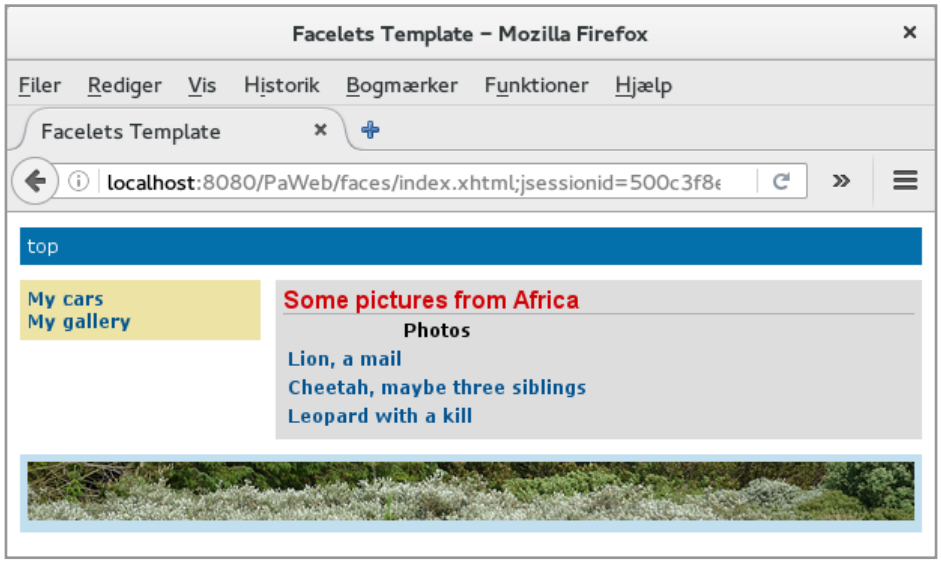
```
 public GalleryController()
 {
  photos.add(new Photo("lion", "Lion, a mail"));
  photos.add(new Photo("cheetah", "Cheetah, maybe three siblings"));
  photos.add(new Photo("leopard", "Leopard with a kill"));
 }

 public List<Photo> getPhotos()
 {
  return photos;
 }
}
```

It is very simple controller with a single property, which is a list of *Photo* objects. Then I have changed *gallery.xhtml* so it now inserts code in the center area:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//
EN" … > <html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
```

```
<body>
 <ui:composition template="./mainTemplate.xhtml">
  <ui:define name="top">
   top
  </ui:define>
  <ui:define name="content">
   <h:form>
    <h1>Some pictures from Africa</h1>
    <h:dataTable value="#{galleryController.photos}" var="img">
     <f:facet name="header">Photos</f:facet>
     <h:column>
      <h:commandLink value = "#{img.text}" action = "photoview">
       <f:param name = "imagename" value = "#{img.name}.jpg" />
      </h:commandLink>
     </h:column>
    </h:dataTable>
   </h:form>
  </ui:define>
 </ui:composition>
</body>
</html>
```

The content is a form with a *dateTable* element, and the table contains a number of *commandLink* elements determined by the controller class. It is an example of an action with a parameter, where the parameter is defined by a *param* element. If you then try the program, you get the following window



The individual *commandLink* elements refer to a view that has not yet been created. First, I have copied three more pictures to the *resources/images* folder (*lion.jpg*, *leopard.jpg* and *cheetah.jpg*) and then created a JSF page called *photoview.xhtml* to display the image. Note that each *commandLink* has a parameter named *imagename* and the value is the name of the image:

```
<f:param name = "imagename" value = "#{img.name}.jpg" />
```

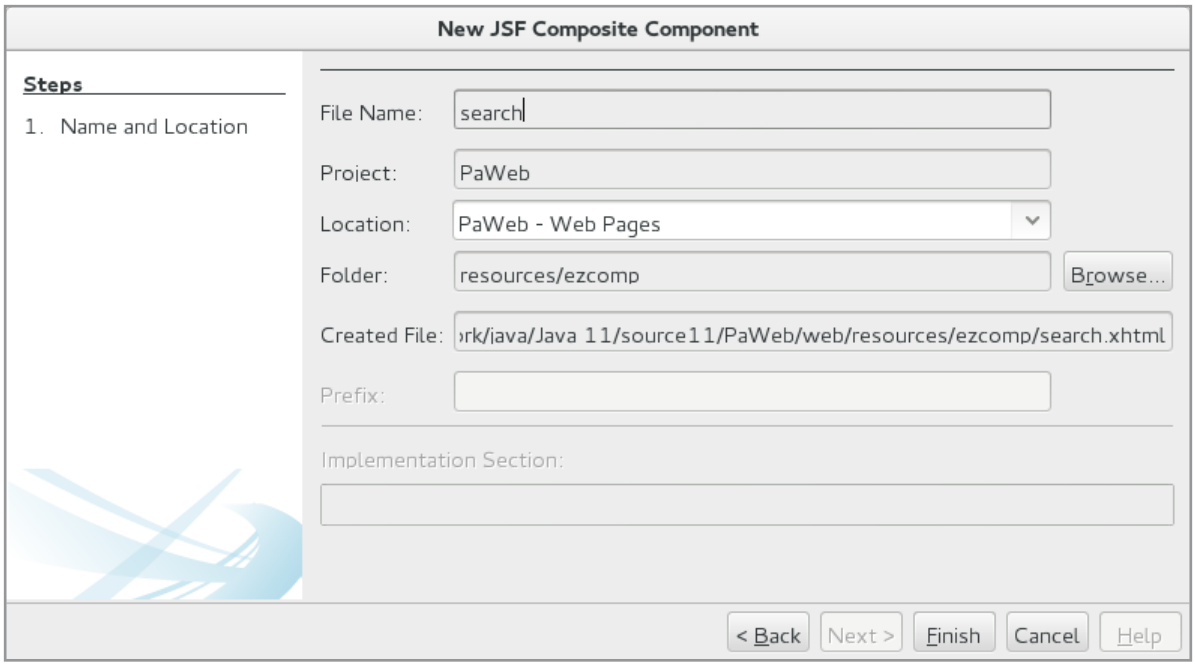
Then there is *photoview.xhtml*:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
 <body>
  <ui:composition template="./mainTemplate.xhtml">
   <ui:define name="content">
    <h2>This is one of my pictures</h2>
    <h:graphicImage library="images" style="width: 800px; height: 450px"
     name="#{request.getParameter('imagename')}"/>
   </ui:define>
  </ui:composition>
 </body>
</html>
```

It is also a view that uses the template, but it only defines the code for the center section. Here you insert an image and you should primarily note how to refer to the image's name, which is the parameter transferred from *gallery.xhtml*.

If you then try the application and click on one of the three links, opens a page that shows an image. Note that the page shows the correct top section as defined in the template. Note that you get back to the gallery page by clicking the link in the menu on the left.

The last thing missing about the gallery is the top section where it should add code to this section. The goal is to show how to create and add composite component. It is a file of JSF elements that can be inserted in another view and possibly in multiple views. In JavaServers Faces, I have chosen *JSP Composite Component*:



Here I enter the name (in this case *search*) and where the component is to be placed. NetBeans suggests *resources/ezcomp*, and it is recommended to keep this name. When you click *Finish*, NetBeanns creates a file *search.xhtml*:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:cc="http://xmlns.jcp.org/jsf/composite">
 <!-- INTERFACE -->
 <cc:interface>
 </cc:interface>
 <!-- IMPLEMENTATION -->
 <cc:implementation>
 </cc:implementation>
</html>
```

It is nothing but a simple skeleton, in which an interface part must define the component's attributes, while in the implementation section it is necessary to define the code to be rendered by the browser. The finished component is shown below:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:cc="http://xmlns.jcp.org/jsf/composite">
 <cc:interface>
  <cc:attribute name="searchList"/>
  <cc:attribute name="searchAction"
   default="#{searchController.searchData(cc.attrs.searchList)}"
   method-signature="java.lang.String action(java.util.List)"/>
 </cc:interface>
 <cc:implementation>
  <h:form id="searchForm">
   <h:outputText id="error" value="#{searchController.errorText}"/>
   <br/>
   <h:inputText id="searchText" styleClass="searchBox" style="width: 200px"
    value="#{searchController.searchText}"/>
   <h:commandButton id="searchButton" value="Search"
    action="#{cc.attrs.searchAction}" />
   <h:commandLink value="#{searchController.foundText}" action="photoview">
    <f:param name = "imagename" value = "#{searchController.imageText}.jpg" />
   </h:commandLink>
  </h:form>
 </cc:implementation>
</html>
```

Note, first, that I have added the two default namespaces regarding JSF elements. Then, two attributes are defined, the first being a simple attribute called *searchList*. The other is called *searchAction* and defines a method. It has a default value which is a method defined in a named bean named *SearchController* that has the value of the *searchList* attribute as a parameter. In addition, there is a definition of the method's signature as a method that returns a *String* and has a *List* as a parameter. The implementation section starts with an *outputStream*, to be used for an error message from *serachController* (if you are looking for something that does not exist). Next, there is an *inputText* where the user can enter a search text and subsequently a *commandButton* whose action is the method defined in the interface section. Finally, there is a *commandoLink* whose value is bound to a property in *searchController* and whose action is *photoview.xhtml*. This link defines a parameter that is also bound to a property in *searchController*.

As can be seen, this composite component uses a named bean:

```
package paweb.beans;

import java.util.*;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
import java.io.Serializable;

import paweb.models.*;

@Named(value = "searchController")
@SessionScoped
public class SearchController implements Serializable
{
 private String searchText;
 private String errorText;
 private String imageText;
 private String foundText;

 public SearchController()
 {
 }

 public void searchData(List<Data> list)
 {
  errorText = "";
  for (Data data : list)
   if (data.getText().toLowerCase().contains(searchText.toLowerCase()))
   {
```

```
    imageText = data.getName();
    foundText = data.getText();
    return;
   }
 imageText = "";
 foundText = "";
 setErrorText("Not found");
}

public String getSearchText()
{
 return searchText;
}

public void setSearchText(String searchText)
{
 this.searchText = searchText;
}

public String getErrorText()
{
 return errorText;
}
```

```
 public void setErrorText(String errorText)
 {
  this.errorText = errorText;
 }

 public String getImageText()
 {
  return imageText;
 }

 public String getFoundText()
 {
  return foundText;
 }
}
```

It defines four properties and must be used to search for images. The four properties are used for:

1. *searchText,* like what is being searched for
2. *errorText*, which is an error message in the case that nothing are found
3. *imageText*, which is the text of the object found
4. *foundText*, which is the name of the object found

Otherwise, the main method is *searchData()* as the method called from the composite component. The method has a parameter that is a list of objects of the type *Data*, which is the list to be searched. Finding an object initializes the two properties *imageText* and *foundText* and otherwise *errorText* is assigned a value.

After the component is complete, it must be used in *gallery.xhtml*:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ez="http://xmlns.jcp.org/jsf/composite/ezcomp">
 <body>
  <ui:composition template="./mainTemplate.xhtml">
   <ui:define name="top">
    <h2>Pouls website</h2>
    <ez:search id="searchGallery" searchList="#{galleryController.photos}"/>
   </ui:define>
```

```
   <ui:define name="content">
    <h:form>
     <h1>Some pictures from Africa</h1>
     <h:dataTable value="#{galleryController.photos}" var="img">
      <f:facet name="header">Photos</f:facet>
      <h:column>
       <h:commandLink value = "#{img.text}" action = "photoview">
        <f:param name = "imagename" value = "#{img.name}.jpg" />
       </h:commandLink>
      </h:column>
     </h:dataTable>
    </h:form>
   </ui:define>
  </ui:composition>
 </body>
</html>
```

Here you should note how to use the component and here specifically how to specify the list to be searched using the attribute defined in the component's interface part.

Then the first part of the application is completed and I have to implement the part corresponding to the first menu item in the left sector. It is in principle a repeat of much of the above, but the following is made.

I have created a model class completely identical to *Photo*:

```
package paweb.models;

public class Car extends Data
{
 public Car(String name, String text)
 {
  super(name, text);
 }

 public String toString()
 {
  return getText();
 }
}
```

There is of course no reason for this class, and it is included only to show that in practice, the contents of this class would be different from *Photo*.

A *CarController* class has also been added, which corresponds to *GalleryController*, and the two classes are basically the same, so I do not want to show the code here. A *FaceLet carview.xhtml* has also been added, which is almost identical to *photoview.xhtml*. Then there is a FaceLet *cars.xhtml*, but before I look at it, I will add a small change to the composite component:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml" … >
 <cc:interface>
  <cc:attribute name="searchList"/>
  <cc:attribute name="searchAction"
   default="#{searchController.searchData(cc.attrs.searchList)}"
   method-signature="java.lang.String action(java.util.List)"/>
  <cc:attribute name="viewer"/>
  <cc:attribute name="showAction"
   default="#{searchController.show(cc.attrs.viewer)}"
   method-signature="java.lang.String action(java.lang.String)"/>
 </cc:interface>
 <cc:implementation>
  <h:form id="searchForm">
   <h:outputText id="error" value="#{searchController.errorText}"/>
   <br/>
   <h:inputText id="searchText" styleClass="searchBox" style="width: 200px"
```

```
    value="#{searchController.searchText}"/>
   <h:commandButton id="searchButton" value="Search"
    action="#{cc.attrs.searchAction}" />
   <h:commandLink value="#{searchController.foundText}"
    action="#{cc.attrs.showAction}">
    <f:param name = "imagename" value = "#{searchController.imageText}.jpg" />
   </h:commandLink>
  </h:form>
 </cc:implementation>
</html>
```

I have expanded with two new attributes, one of which refers to a method in *SearchController*. The goal is to parameterize the action to be performed if you click on the link. The method in *SearchController* is

```
public String show(String name)
{
 return name;
}
```

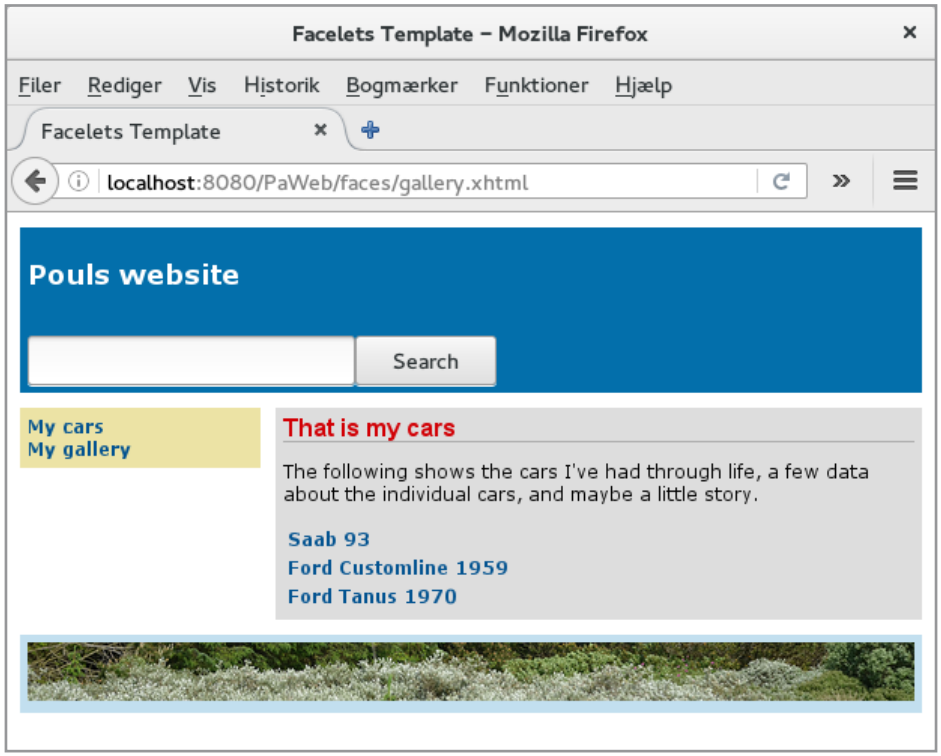
After this change, the *gallery.xhtml* view must also be updated to initialize the component's attribute. Finally, *cars.xhtml* can be written as follows:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ez="http://xmlns.jcp.org/jsf/composite/ezcomp">
 <body>
  <ui:composition template="./mainTemplate.xhtml">
   <ui:define name="top">
    <h2>Pouls website</h2>
    <ez:search id="searchGallery" searchList="#{carController.cars}"
     viewer="carview"/>
   </ui:define>
   <ui:define name="content">
    <h:form>
     <h1>That is my cars</h1>
     <p>The following … </p>
     <table>
      <ui:repeat var="car" value="#{carController.cars}">
       <tr>
        <td>
        <h:commandLink value = "#{car.text}" action = "carview">
```

```
          <f:param name = "imagename" value = "#{car.name}.jpg" />
         </h:commandLink>
         </td>
        </tr>
       </ui:repeat>
      </table>
     </h:form>
    </ui:define>
  </ui:composition>
 </body>
</html>
```

Note that another loop has been used. There is no particular reason why, in addition, if you want to take care of how the table is built, for example because of styles.

Finally, the directorate *resources/images* are updated with three images, and the application is executed, and if you choose the left menu *My cars*, you get the window:

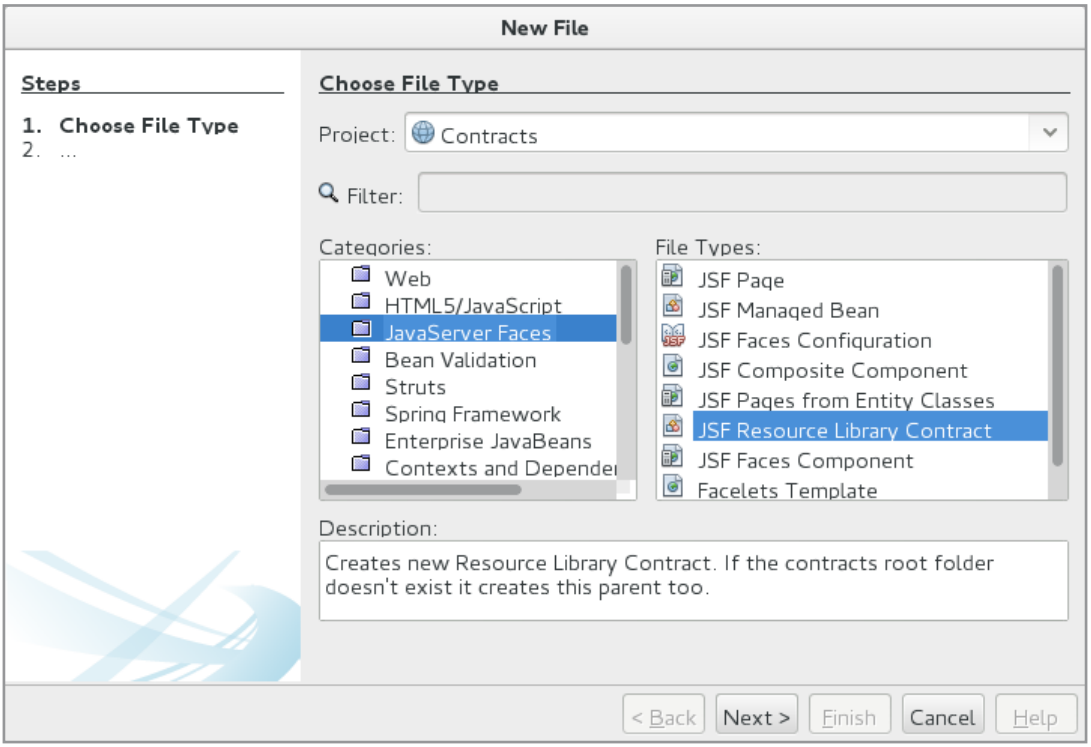


6.4 THEMES

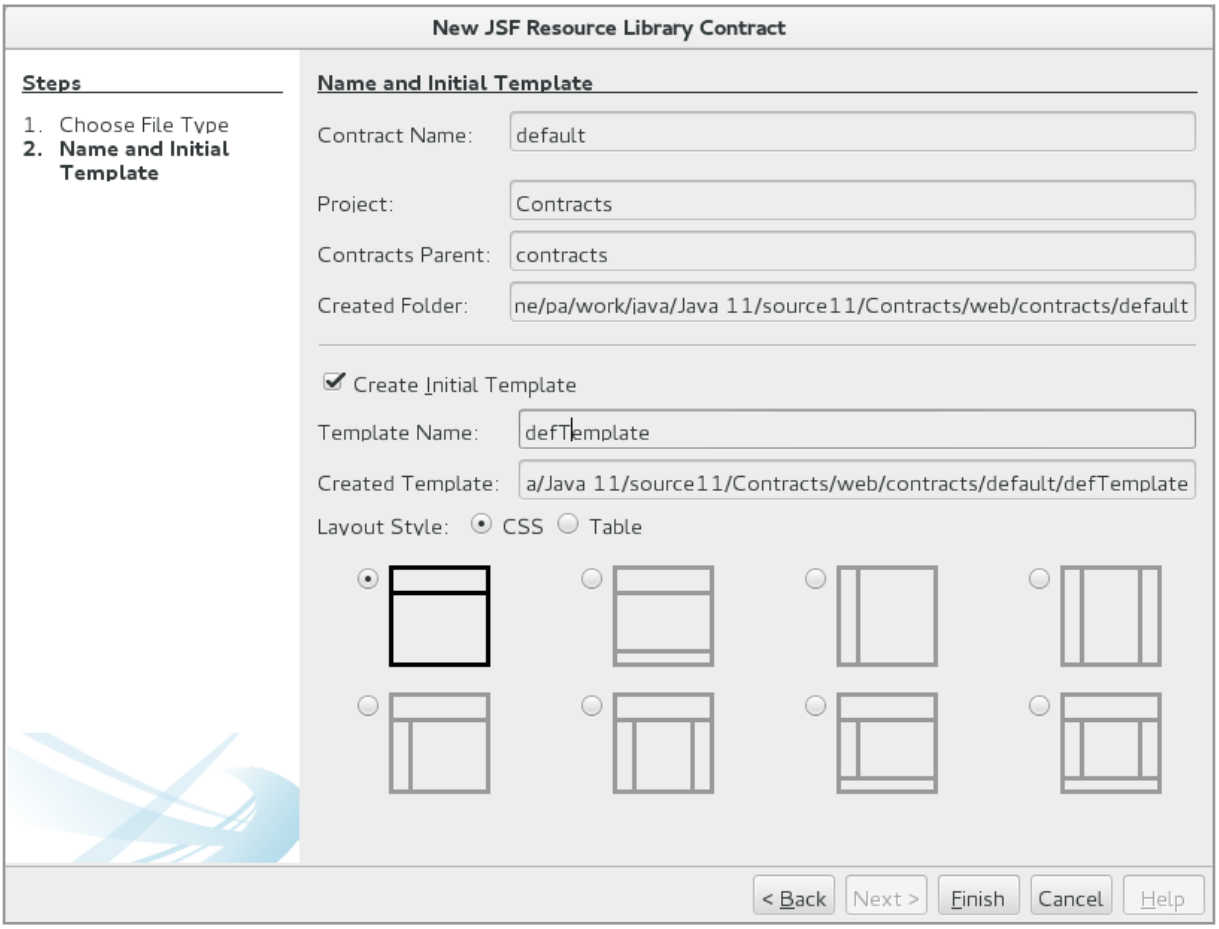

Facelets Template – Mozilla Firefox
Filer Rediger Vis Historik Bogmærker Funktioner Hjælp
Facelets Template
localhost:8080/Contracts/faces/index.xhtml
Select a theme
Default
Select
There was a man named Ulf, son of Bjalf, and Hallbera, daughter of Ulf the fearless; she was sister of Hallbjorn Half-giant in Hrafnista, and he the father of Kettle Hæing. Ulf was a man so tall and strong that none could match him, and in his youth he roved the seas as a freebooter. In fellowship with him was one Kari of Berdla, a man of renown for strength and daring; he was a Berserk. Ulf and he had one common purse, and were the dearest friends.

If you have a web application, it is possible to apply themes where the application's views are rendered corresponding to which theme is selected. For example, an application may appear differently depending on which user is logged in. Themes are based on templates, and I will in this section show how it works. If you open the *Contracts* application, you get the window above. That is, a simple window with a text. If you in the drop down list select the *dark* theme and click *Select*, the window changes to:

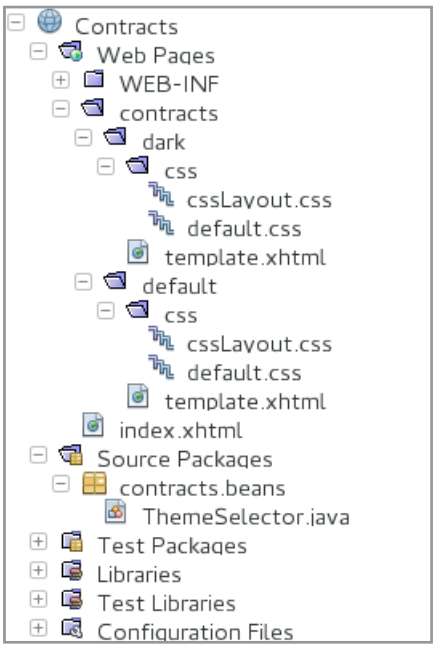
The starting point is a usual JSF application called *Contracts*. In addition, I have added a *Resource Library Contract*:



and when clicking *Next*, I have assigned a name (*default*) and selected a template (named *template* see below). NetBeans then creates a directory *contracts/default* with a template (and two style sheets). I have then repeated it all and created another *Resource Library Contract*, but this time with the name *dark*. Otherwise there are no differences.



After I've created the two contracts, the content of the project is following, where there is also a named bean named *ThemeSelector* added:



The two templates are the same, but the idea is that they can be modified as desired. This is not the case in this case, but the content is:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" ,,, >
<html xmlns="http://www.w3.org/1999/xhtml" … >
  <h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <h:outputStylesheet library="css" name="default.css"/>
    <h:outputStylesheet library="css" name="cssLayout.css"/>
    <title>Facelets Template</title>
  </h:head>
  <h:body>
    <div id="top" class="top">
      <ui:insert name="top">Top</ui:insert>
    </div>
    <div id="content" class="center_content">
      <ui:insert name="content">Content</ui:insert>
    </div>
  </h:body>
</html>
```

I have then modified *index.xhtml* to (altermnatively you could have created a new *FaceLets Template Client*):

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns=" … >
 <h:body>
  <f:view contracts="#{themeSelector.name}">
   <ui:composition template="/template.xhtml">
    <ui:define name="top">
     <h:form>
      <h:outputLabel for="selector" value="Select a theme"/>
      <h:selectOneMenu id="selector" value="#{themeSelector.name}">
       <f:selectItem itemLabel="Default" itemValue="default"/>
       <f:selectItem itemLabel="Dark" itemValue="dark"/>
      </h:selectOneMenu>
      <h:commandButton value="Select" action="index"/>
     </h:form>
    </ui:define>
    <ui:define name="content">
     <p> … </p>
    </ui:define>
   </ui:composition>
  </f:view>
 </h:body>
</html>
```

Here you should note that it is all enclosed by a *f:view* element, as determined by the bean object, determines which theme to use. Otherwise, the rest is simply code inserted into the two sections that the template defines. Also note that the *commandButton* element at the top performs an submit to the page itself, which means that the page is rendered based on which theme is selected.

Using themes is quite simple, but for the time being, the two themes are the same and there will be no difference if you choose one or the other theme. This is solved by changing the two templates as well as the style sheets they use. I have changed the template for the default theme to the following:

```
<h:body>
 <div id="top" class="top">
  <ui:insert name="top">Top</ui:insert>
 </div>
 <div id="content" class="center_content">
  <h2>The theme is default</h2>
  <ui:insert name="content">Content</ui:insert>
 </div>
</h:body>
```

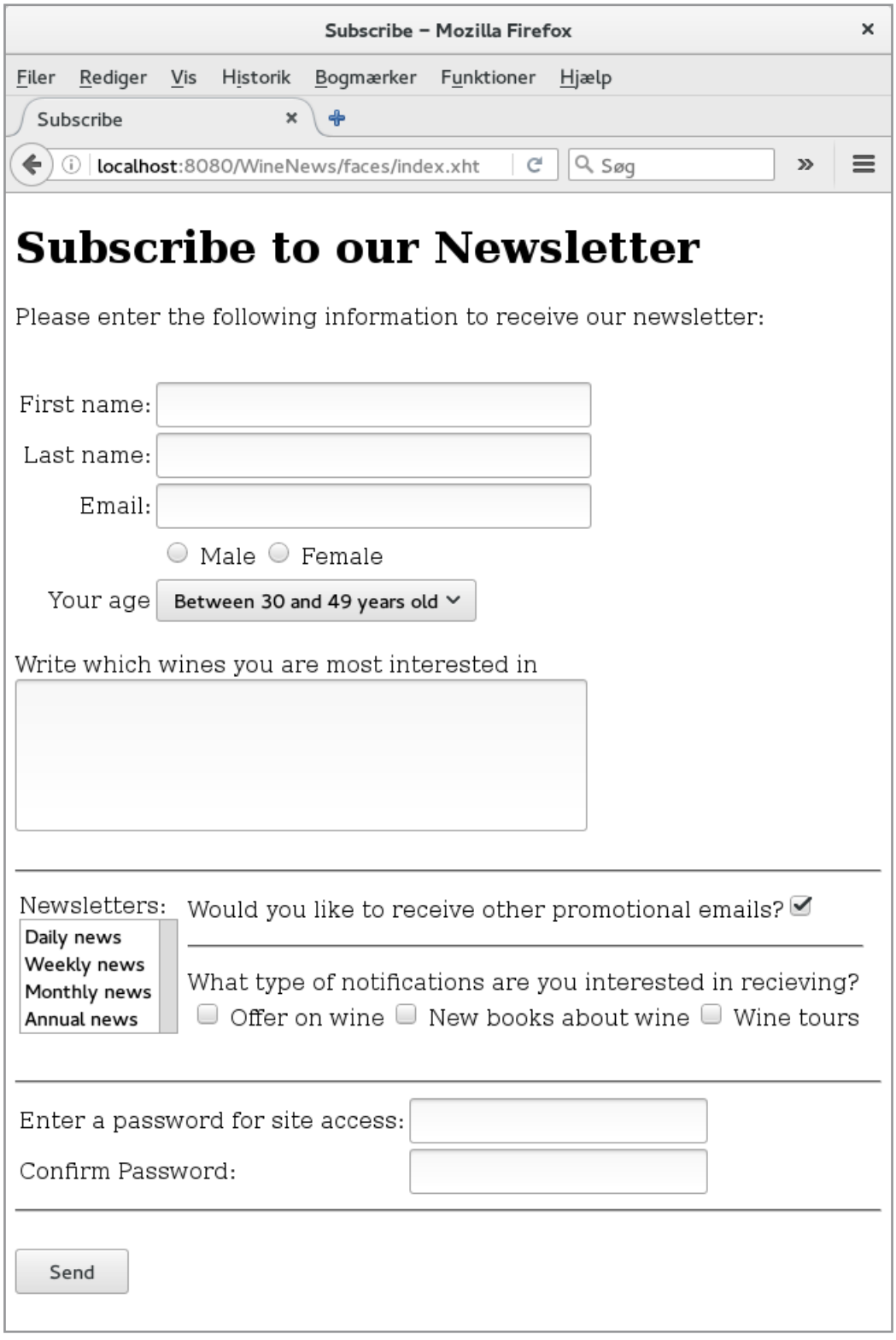
And the other template has been changed accordingly, just so the text is *The theme is dark*. Finally, for the theme dark, I changed a single claas in *cssLayout.css* to

```
.center_content {
  position: relative;
  background-color: #444444;
  color: #ffffff;
  padding: 5px;
}
```

PROBLEM 3

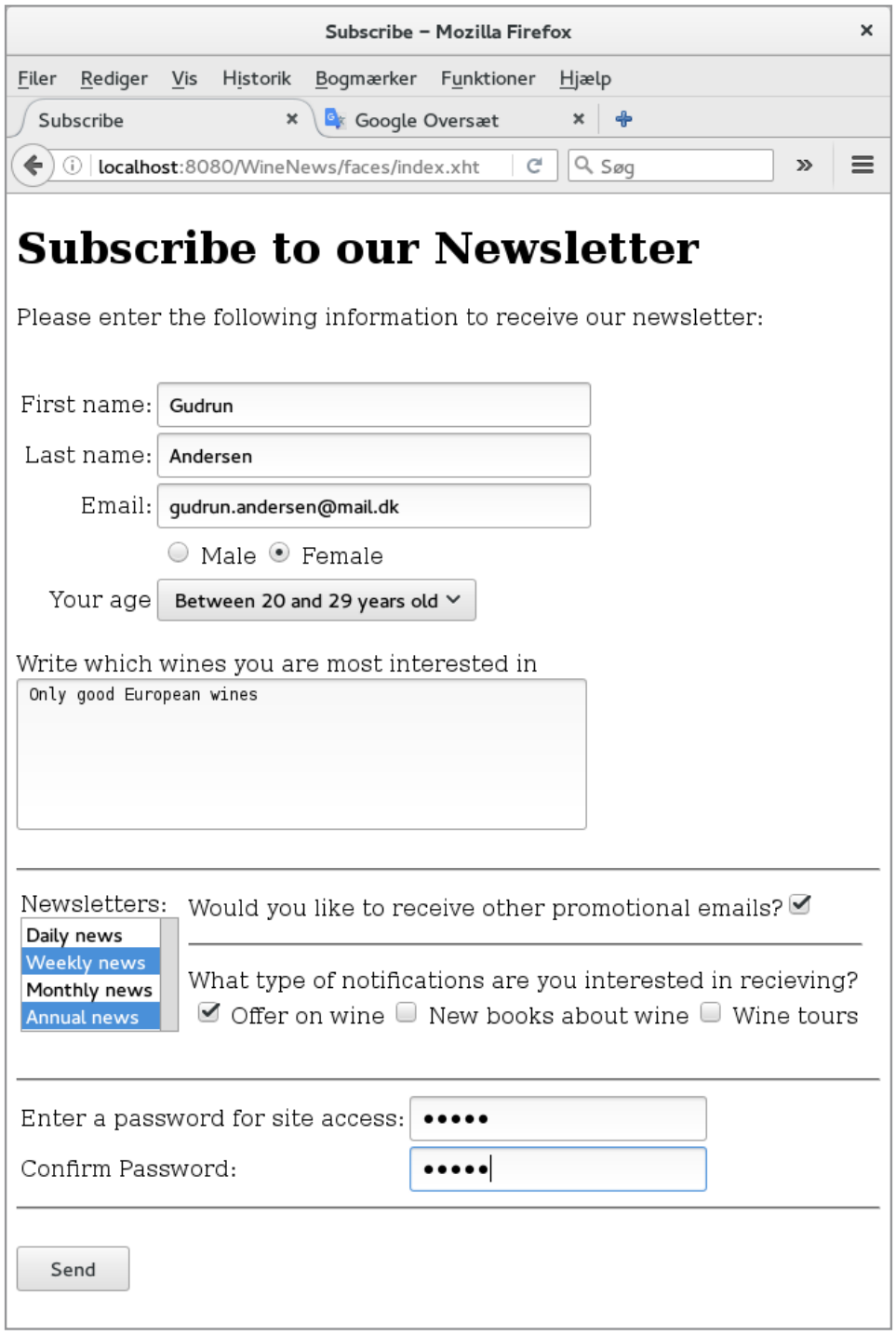
In this task you must write a web application that opens a simple welcome page:

The content is not very important, but there must be a link to the form below. It should illustrate that a person wishes to subscribe to newsletters (in this case newsletters from a wine club or equivalent). The goal is to work with JSF elements, which I have not mentioned. It is part of the task to find out what the individual elements are called and how they are used.

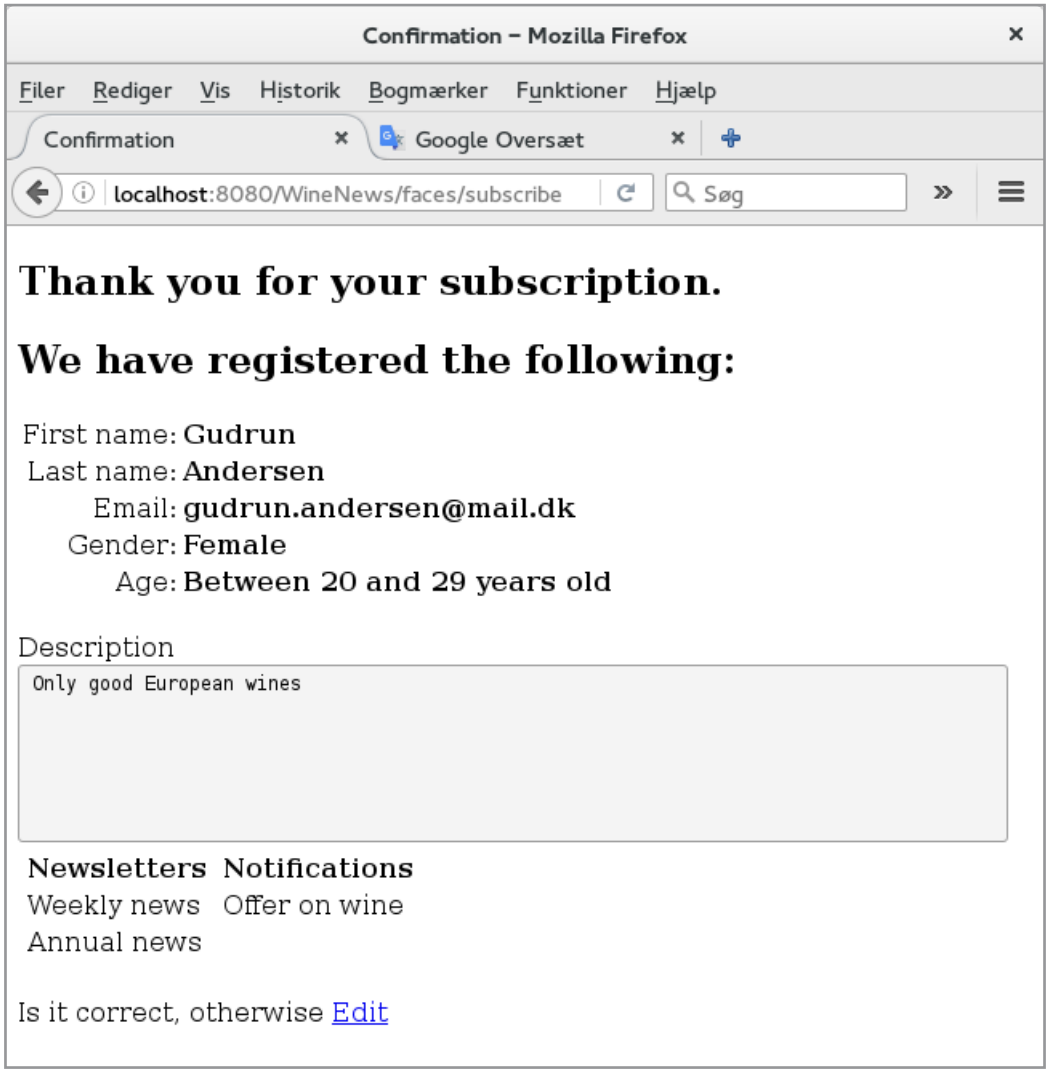


When the form is sent to the server, it must be validated that some first name and last name has been entered, and a legal email address has been entered. It must also be validated that gender has been chosen. Finally, it must be validated that the contents of the two password fields are the same.

Below is an example of a completed form:



If you click *Send*, you will receive the confirmation page below, where you can click *Edit* to change the entered data:



6.5 UPLOAD IMAGES

To finish this chapter and *JavaServer Faces* I will show a small example of a web application, where you can upload images to a web server. In fact, it is a common problem, so this example, but it is quite easy. The example consists only of *index.xhtml* and a single bean:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" … >
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
 <h:head>
  <title>Upload image</title>
 </h:head>
```

```
 <h:body>
  <h:form enctype="multipart/form-data">
   <h2>Choose a file to upload to the server:</h2>
   <h:inputFile value="#{uploadController.file}">
    <f:ajax listener="#{uploadController.save}" />
   </h:inputFile>
  </h:form>
 </h:body>
</html>
```

It's all done with a single JSF element called *h:inputFile*, which binds to a file property in a bean. With this component you can browse the local file system and upload the image (and even any file). Now, the image must also be saved, which happens with an *ajax* command. *Ajax* is dealt with in the next book, and here you only have to take note of the syntax, but the result is, that a method is called in the bean class, which is then responsible for saving the image. The bean class is as follows:

```
package upload.beans;

import java.io.*;
import java.nio.file.*;
import javax.inject.Named;
import javax.enterprise.context.*;
import javax.servlet.http.Part;

@Named(value = "uploadController")
@SessionScoped
public class UploadController implements Serializable
{
 private Part file;
 private String uploads = System.getProperty("user.home") + "/tmp";

 public UploadController()
 {
 }

 public Part getFile()
 {
  return file;
 }

 public void setFile(Part file)
 {
  this.file = file;
 }

 public void save()
 {
  try (InputStream input = file.getInputStream())
  {
   Files.copy(input, new File(uploads, file.getSubmittedFileName()).toPath());
  }
  catch (IOException ex)
  {
  }
 }
}
```

A problem associated with the application that needs to be solved is where the images (files) must be saved. Obviously, the application must know where and there must be a directory on the server that is witeable. In this example, a directory is selected under my home directory, but in practice, of course, another solution is required. For example, you can study how the problem is solved in the final example.

The class has a variable named *file*, and here you should especially note the type:

```
javax.servlet.http.Part
```

Otherwise, there is not much to notice besides the method *save()*, which creates a copy in the selected directory of the uploaded file.

7 A LAST EXAMPLE

The task is to write a web application for a veteran car club where the club members can presents their cars to the public. Thus, it is a typical web application that has two presentations:

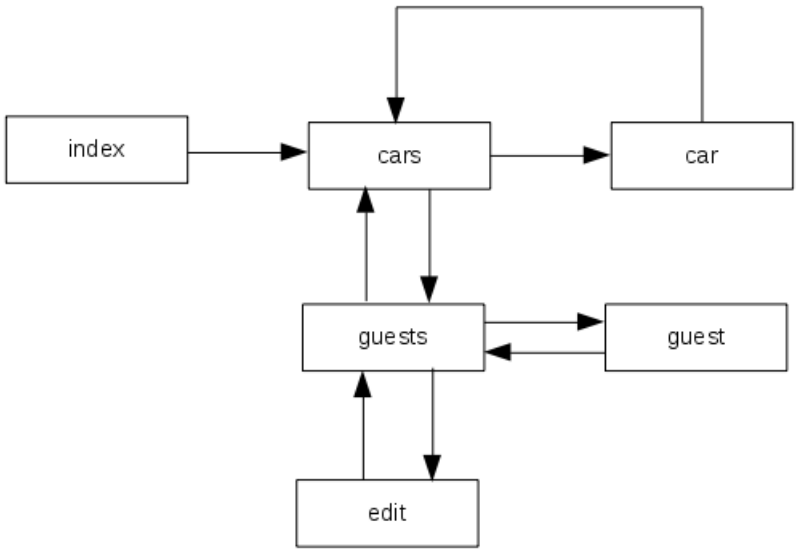
1. the public page, which is the actual website and can be used by all users
2. an administration page that can only be used by the person (s) that has to administer the site (web masters), and this part of the application will usually require logon

In this case there must be one or more web masters who can all and especially maintain information about club members. The club's other members should also be able to use the administration part, but should only be able to maintain information about their own cars.

With regard to the public side, everyone must have access to it, and the site should basically provide information about members' cars and typically one or more pictures. In addition, the site must be attached to a guestbook, where everyone has the opportunity to attach a comment.

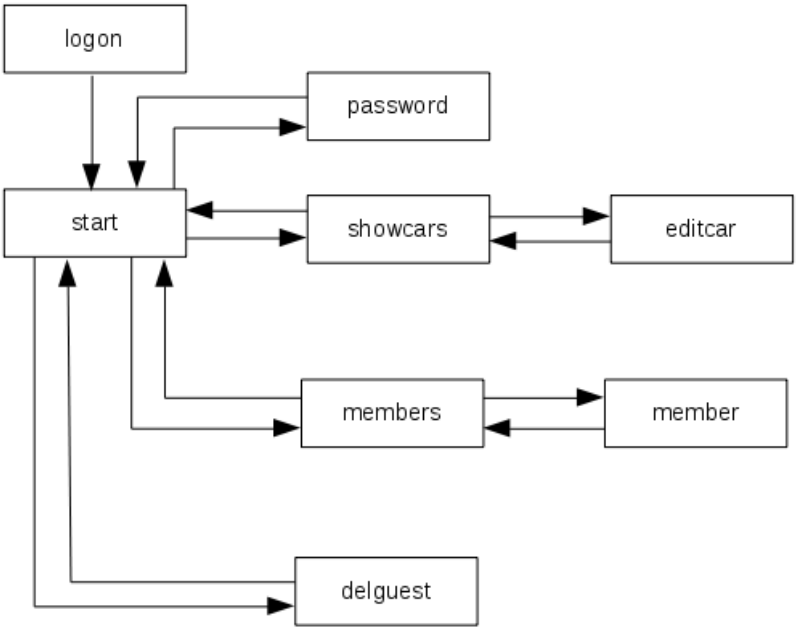
7.1 ANALYSIS

The public page is generally simple and consists of the following views, where the arrows show how to navigate the site:



1. *index* is a home page or welcome page that gives a brief presentation of the site
2. *cars* is a search page where the user has the opportunity to search among members cars
3. *car* is a presentation page that shows details about a single car, including information about the car's owner
4. *guests* is the guest book where you get a table of contents sorted by date
5. *guest* shows what a particular guest has written
6. *edit* is a page where the user can write in the guestbook

With regard to the administration part, it must include the following pages:



1. *logon* is a logon page for webmasters and members
2. *start* is the admin part's home page
3. *password* is used to change password
4. *showcars* shows an overview of the individual members cars and for webmasters all cars
5. *editcar* is used to edit information about a car, including to create a new car, and to delete an existing car
6. *members* shows an overview of club members – the function can only be used by webmasters
7. *member* used to edit information about a member including creating a new member as well as deleting an existing member – the feature can only be used by webmasters
8. *delguest* used to delete pages in the guestbook if something has been written, that the club will not accept – the function can only be used by webmasters

The webmaster can specifically maintain cars not owned by a member. If so, it should appear on the public page.

Data dictionary

Members:

- number (which is a unique 5 digit integer)
- name
- address
- zip code
- mail (that is used as user name)
- phone
- password
- description
- user role (0 = default web master, 1 = web master, 2 = member)

Default web master is a standard web master (a superuser) created when the database is created and can not be deleted. Other webmasters are, in principle, just a common member, but with administrator rights.

Description is intended as a possibility that a member can give a presentation of himself or his cars.

The guestbook:

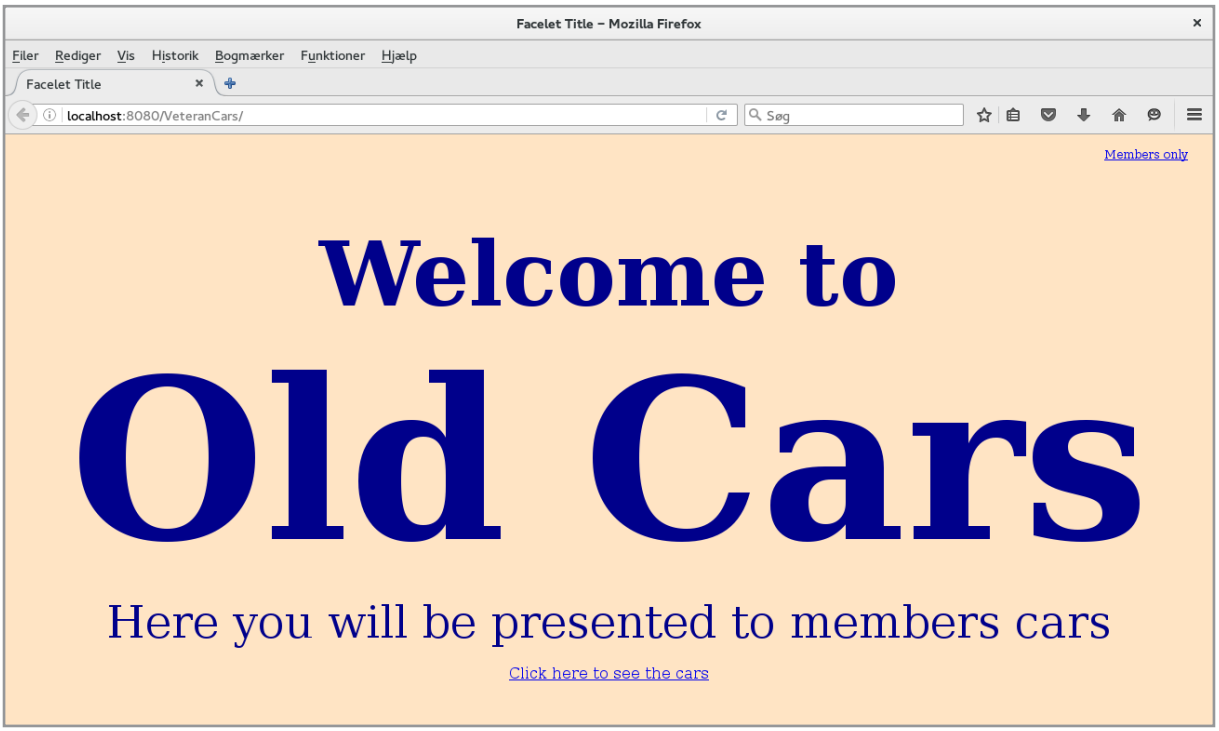
- name
- zip code
- mail
- date (when it is written in the guestbook)
- text

Cars:

- country code (where the car is produced)
- producer (for example Opel, Ford, …)
- name
- model designation
- variant designation within the model
- motor (for example V8)
- effect (motor effect in HK and possible unit)
- production year
- owner (member)
- description
- picture (one or more pictures)

7.2 DESIGN

The *index.xhtml* page shows the following view, which is a simple welcome page. At the bottom there is a link to the public page and thus to *cars.xhtml*, while in the upper right corner there is a link to the administration part and thus a link to *login.xhtml*:



The application's data must be stored in a database, except for images uploaded to a directory on the server. The database should primarily have three tables:

1. to members
2. to cars
3. to the guestbook

and the complete database design is shown in the following script:

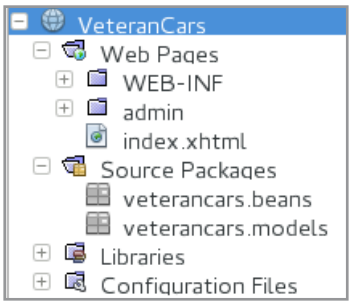
```
use sys;
drop database if exists veteran;
create database veteran;
use veteran;

create table guest (
 id int auto_increment primary key,
 name varchar(100),
 code char(4),
 mail varchar(50),
 date date,
 text text
);
```

```
create table owners (
 id char(5) primary key,
 name varchar(100),
 address varchar(100),
 code char(4),
 mail varchar(50),
 phone varchar(20),
 text text,
 role int default 1,
 passwd varchar(150)
);

create table brands (
 id int auto_increment primary key,
 country char(2),
 name varchar(100)
);

create table cars (
 id int auto_increment primary key,
 brand int,
 name varchar(100),
 model varchar(100),
```

```
 variant varchar(100),
 motor varchar(100),
 effekt varchar(20),
 year int,
 owner char(5),
 text text,
 foreign key (brand) references brands (id),
 foreign key (owner) references owners (id)
);

create table image (
 id int auto_increment primary key,
 name varchar(100),
 car_id int,
 foreign key (car_id) references cars(id)
);

insert into owners (id, name, mail, text, role) values
("00000", "Web master", "poul.klausen@mail.dk", "Super User", 0);
```

The table image contains names of images (relative to the image directory). The reason for this table is that more images may be attached to the same car. Similarly, there is a table of brands that contain a list of car brands. The reason for this table is partly that many cars have the same manufacturer, and that the car brand is to be used when searching cars.

The overall design is as follows:



where the directorate *admin* must contain all FaceLets regarding the administration part.

A particular issue is security and how to ensure that unauthorized persons can not change the content of the site's database. In this case, security must be given a low priority. Partly because the content of the page does not require special protection and one can not expect anyone to use energy to attack the page, and partly because (and primarily) that I have not yet addressed how security issues are addressed. To access the administration section, a member (car owner) must enter his membership number and password. It must be the responsibility of the member to create an appropriate password and it is stored encrypted in the database. In addition to not claiming the strength of the password and that it is the member's responsibility to create a password, the biggest weakness is that the password is sent in clear text (unencrypted) from client to server, and can thus be captured by an attacker .

7.3 PROGRAMMING

The finished site consists of 15 facelets and additionally controllers (named beans), model classes and other auxiliary classes. There is no full match between the 15 facelets and the above navigation charts (the diagrams show only 14 views), and typically changes will be made during the programming as to what views the completed solution will contain. Nevertheless, I find it helpful to use the time to draw up the charts, even if the final result differs, and the more careful you are in the analysis, the greater the chance that programming means only minor adjustments.

Compared to many practical web applications, the current application is small and I will generally not display the code for either facelets or Java classes, but I will explain how the application is developed. However, you are encouraged to study the code, and as compared to what is otherwise dealt with regarding web applications in this book, many details are solved.

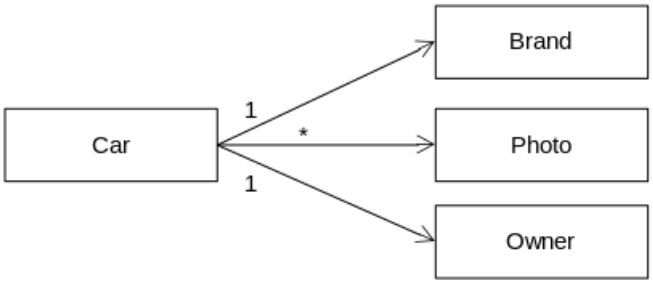
The administration part

I will start with the management part what is typical for developing web applications of this type. The public page can only be developed after adding data to the database.

To the model layer is added the following classes:

- Brand
- Car
- Photo
- Owner
- Guest

which models the database's tables. It's all simple model classes, maybe close to *Car*, which contain references to other objects:



In addition, there is the class *Repository*, which is a singleton with methods for maintaining the database. The class is relatively complex (comprehensive), but does not contain anything new compared to corresponding classes from other programs. Parameters for the database (username and password) are stored in *web.xml*, as well as the name of the directory that contains the uploaded images:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app … >
 ….
 <welcome-file-list>
```

```
    <welcome-file>faces/index.xhtml</welcome-file>
  </welcome-file-list>
  <listener>
   <listener-class>veterancars.models.ParamsFactory</listener-class>
  </listener>
  <context-param>
   <param-name>usr</param-name>
   <param-value>pa</param-value>
  </context-param>
  <context-param>
   <param-name>pwd</param-name>
   <param-value>Volmer_1234</param-value>
  </context-param>
  <context-param>
   <param-name>images</param-name>
   <param-value>/home/pa/tmp</param-value>
  </context-param>
</web-app>
```

For images, they must be uploaded to a directory of write rights. Each time an image is uploaded, a row is created in the table *image* and the row is assigned an auto number. This number is at the same time used as the image file name, and if, for example, an image is assigned to the number 123 (by the database server), the image is saved under the name *123.jpg*. It is thus decided that the program will only support jpg images (photos).

All views regarding the administration are located in the admin folder, and almost one is associated with a controller. When you select administration from the welcome page, you will see *logon.xhtml*, where you must enter member number and password. Then you will get to the management section's welcome page, called *start.xhtml*. This page has a menu of features at the top. Corresponding to the requirement specification, the administrative part must have the following functions:

1. Maintenance of car brands
2. Maintenance of cars
3. Maintenance of owners (members)
4. Maintenance of the guestbook
5. Change password

The first function is simple and can only be performed by a user with administrator privileges. It includes two views:

1. *brands.xhtml*, which shows an overview of all the car brands that have been created and you are editing a car brand by clicking on it
2. *brand.xhtml* which are used to create and maintain car brands

The next function is the most comprehensive and can be performed by everyone, but if you are not created with administrator right, only your own cars will be displayed. The funtion has three views:

1. *cars.xhtml*, which shows an overview of all cars grouped by the car brand, so if you click on a car brand, you get all the cars for this brand and click on a car, you get all the car details and there is also a link so you can edit the information about the car
2. *car.xhtml*, which is used to create and maintain information about a car
3. *showcar.xhtml*, which shows details about a car and including any pictures

When this feature is complex, it is partly because of uploading of images, where it should be possible to edit the caption as well as *showcar.xhtml* being able to display an image (the contents of a file). The last problem is solved with a servlet.

The third function reminds of the first function and can be performed by all users with administrator privileges. The function has two views:

1. *owners.xhtml*, showing an overview of all owners
2. *owner.xhtml*, used to create and edit owner (member) information

After implementing these functions, I have written the code to the public page.

The public side

From the welcome page you will get to *main.xhtml*, which is the web site's central page. Here you can click on cars using a left-hand menu on where cars are grouped by car brand. You can also search for a car using any text that searches for the car brand, name, model and text. The cars that match are shown in a drop-down box from which you can choose a car.

At the top there is also a link to the guestbook, which consists of two views:

1. *guests.xhtml*, which allows to search the guestbook where you can search by date or enter a search text for search on name and in the text itself
2. *guest.xhtm*, which allows you to write in the guestbook

Completion of the administration part

Lastly, I have completed the administration part. The password change function is simple and consists only of a single view with a controller. The last feature also has just a single view and is virtually identical to *guests.xhtml* and uses the same controller. The only difference is that it is possible to delete a page in the guestbook.

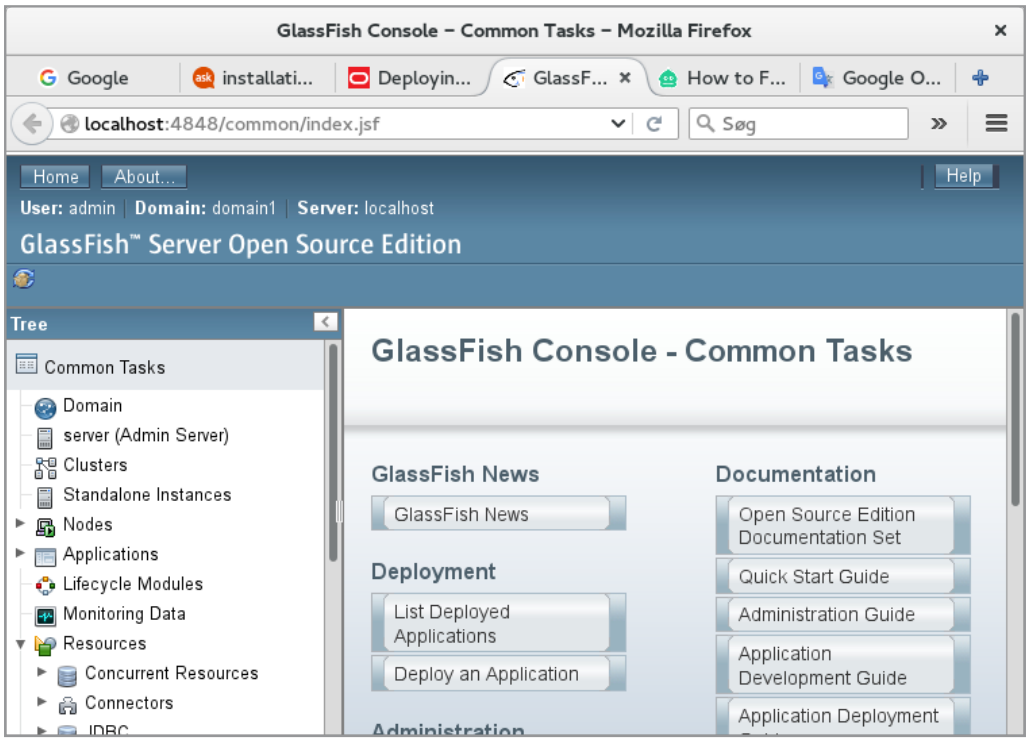
7.4 DEPLOYMENT

In order for a web application to be executed (opens in the browser), the application must be installed on the Glassfish server, a process called deployment. For the time being, I have ignored everything about it, as NetBeans does it quite automatically on a local running Glassfish server. I will now, as an example, show how you can deploy the current application on another machine. The prerequisite is that the machine has a running Glassfish server as well as a running database server. The process requires two files:

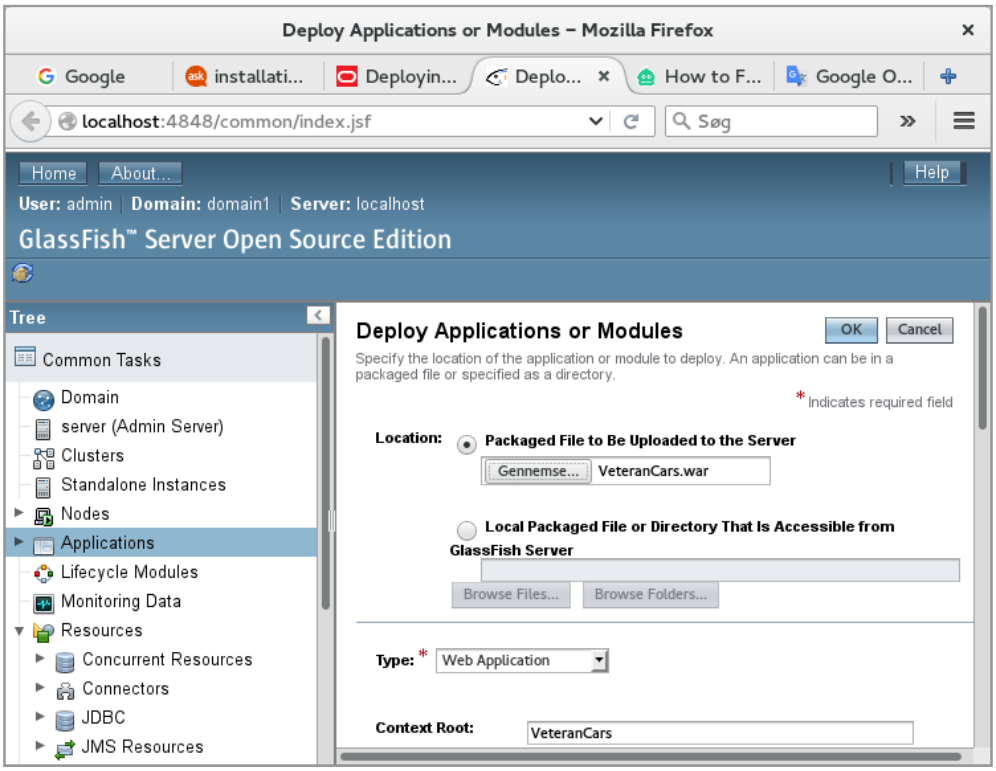
1. *veteran.sql*, which is the SQL script to create the database
2. *VeteranCars.war*, which is the war file with the web application (found in the project's *dist* directory)

The first step is to create the database, which simply consists in running the above script. Next, a directory must be created for images. Note that if you want a database of data, you can backup the database from the developer machine and create a restore of it, while also remembering to copy the images from the developer machine.

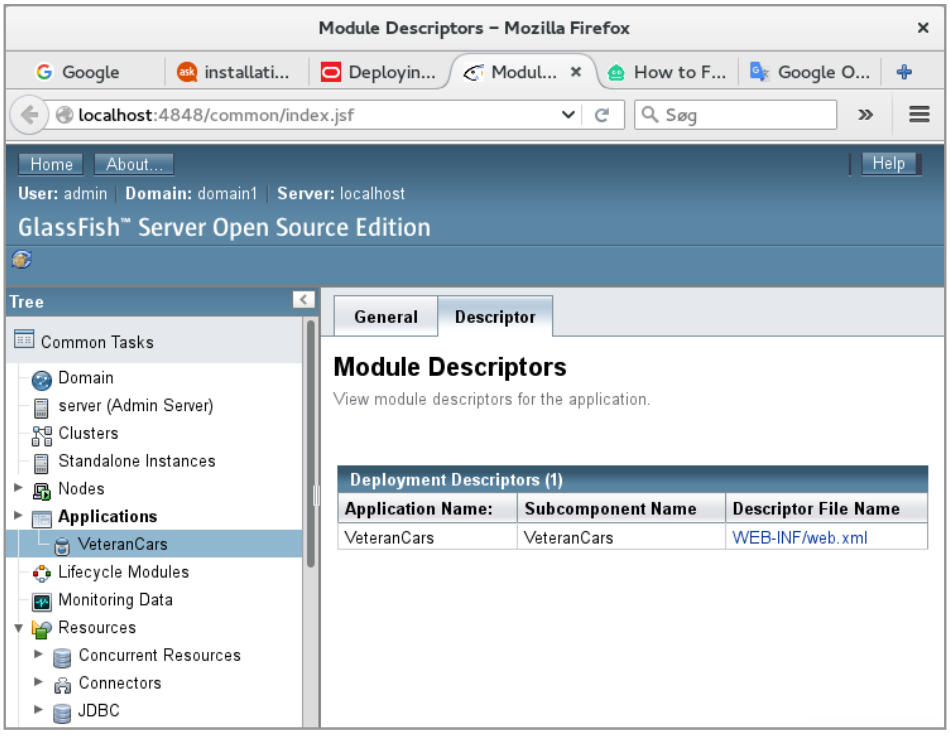
For deployment, open the admin program in the browser:



Here you have to click on *Applications* on the left side and then on the tab *Deploy* … and then you can browse to the war file (see below). All other settings can be retained as default, and then click *OK*, the application will be deployed.



All that remains is to ensure that the parameters of *web.xml* for the database server and the directory for images, respectively, are correct. You can edit the file by clicking on the apllication name and selecting the *Descriptor* tab (see below). Here is a link to *web.xml*, where you can edit the file.



Then everything should be ready and the application can be opened in the browser (on another machine) by typing (assuming that the IP address is correct):

```
http://192.168.1.48:8080/VeteranCars
```

8 A FINAL REMARK

The example in the previous chapter should be perceived as a form of status of this introduction to web applications, and it is also a classic example of a web application. The question is then what's missing before you're fully trained as a web developer – and there's more and more than I get in these books. I would like to end up with a little on the above application, and in particular the shortcomings of the application and the requirements for web applications in practice and thus topics that are dealt with to some degree in the following two books. I will look at the program in relation to the following 5 headings:

1. Maintenance friendly
2. Attractive user interfaces
3. Client side programming
4. Authentication and security
5. Distribueret programmering

Maintenance friendly

In this context, thinking about maintenance, I do not think so much about the program's beans and other Java classes, where there is not much to add to what is said otherwise about classes in the previous books in this series, but in instead, the maintenance of the individual facelets. Let me instantly confirm that the application leaves much to be desired at this point. Perhaps this example is ok, as it is a small application, but a larger application developed in the same way as the above will be hopeless to maintain. The problem is the use of styles, where styles are used in many elements around the individual views. There is nothing wrong with styles and that's exactly the meaning, that the layout and look and feel of a web application's views should be implemented by using styles but it should be done using class's defined in style sheets and not enough with that, it must be based on a well-defined plan for how the user interface should look and work. In the current example, individual items are styled ad hoc as there has been a need. It means partly lacking consistency in styling elements, and partly that it is hopeless (and at least time consuming) to change how the individual elements should appear as you have to go around in all views and find the individual elements.

In fact, I have not mentioned styles or style sheets yet, but just used styles as needed and without comments. This is one of the topics in the following book, where I in generally will focus on client side of web applications.

Attractive user interfaces

Looking at proprietary web applications, one quickly recognizes that much energy has been used for a web application to be great (whatever it is) and user-friendly, so users want to visit the site. How to achieve it is not the subject of these books, and although it is based on individualism and fortunate beliefs, there are theories for how a web site should be developed to make it user-friendly and easy to understand and use. These include the use of colors, font and font size, graphics, and ease of navigating the site.

The current example is missing a lot at that point, and as another problem, the application is hard-coded to a large screen (the individual view elements have a fixed size) that does not work. It can also be mentioned that nothing has been done to ensure that the application is browser independent, which are also factors that must be considered in practice.

Client side programming

Often, you will be interested in something dynamic on the client side and without the need for a request to the server. It requires the browser to execute program code, and here is the preferred *JavaScript*, which is interpreted code. Today's web applications simply can not be developed without the use of client code, and the next book will therefore include an introduction to *JavaScript*, which is a simple programming language with the same syntax as Java.

If you look at what I've mentioned in this chapter, it deals only with the client side and thus the browser. The purpose of the current example is to show programming of the server side, but as shown, there is another side of a web applications that this application either does not use or does not use in a particularly good way. That is the subject of the next book.

Authentication and security

A very central issue in developing web applications is security. As mentioned under the design, it is also a place where this application leaves much to be desired, simply because I have not yet addressed what it takes, but it is also a subject being dealt with later.

Basically, it is about protecting the data that a web application maintains, as well as ensuring that the program code performed on the client's machine can not cause damage. Since everyone can in principle send a request to a web server, it is important to ensure that unauthorized users can not get unwanted access to data, and this is done by, for example, authentication where a user must have a login, and they must also ensure data sent between server and client, which occurs by encryption. It sounds simple, but it's no matter what the many attacks on it solutions also tells. The second issue to ensure that no malicious program code is downloaded to the client's machine is solved using special software (antivirus programs) that monitors the data traffic, but also by ensuring that the code that the browser should perform is such, that it can not cause harm by limiting what the code may be allowed. It's also not easy, and again, there are a lot of examples that client machines have been attacked by unwanted software.

The conclusion is that security associated with web applications plays an increasing and extremely important role. There are, however, a solution, namely, not to use the Internet, but as our world looks today, it's not a viable way. It's a little like that would stop using power, which most probably would also reject as an option. There is nothing but recognition that solutions are to develop secure it solutions – if that is possible?

Distributed programming

A program like *VeteranCars* is not a proprietary program, as it is a program that runs with a database on a web server, but in practice many web applications will be distributed programs. Based on the current program, there was nothing in the way that the Glassfish server and database server could be on two different machines, and for example, the uploaded images could be placed on a third machine. Similarly, with reference to what is otherwise said in this chapter, some of the code could be performed on the clients' machines. In fact, you could think about it and consider the individual Java objects on the server as objects that were instanted and placed around on different machines and lived as services that these machines could then make available to clients such as web servers, but also common client machines with installed stand alone applications or mobile phones.

This means a completely different view of programs and a much greater flexibility where a program performs its work by using services that different machines make available over a network that, in practice, is often the Internet. The development of such applications means new demands for the developers, not to mention security requirements. Development of such programs is the subject of the book Java 13.

The conclusion to all is that the development of web applications and distributed applications requires knowledge of a number of technologies and Java APIs, and a large part of the following books will deal with these things.

APPENDIX A: INSTALLATION OF GLASSFISH

To install NetBeans bundle with Java EE and Glassfish you go to the download page for NetBeans:

```
https://netbeans.org/downloads/
```

Here, select the version in column 2 and download it in the usual way:

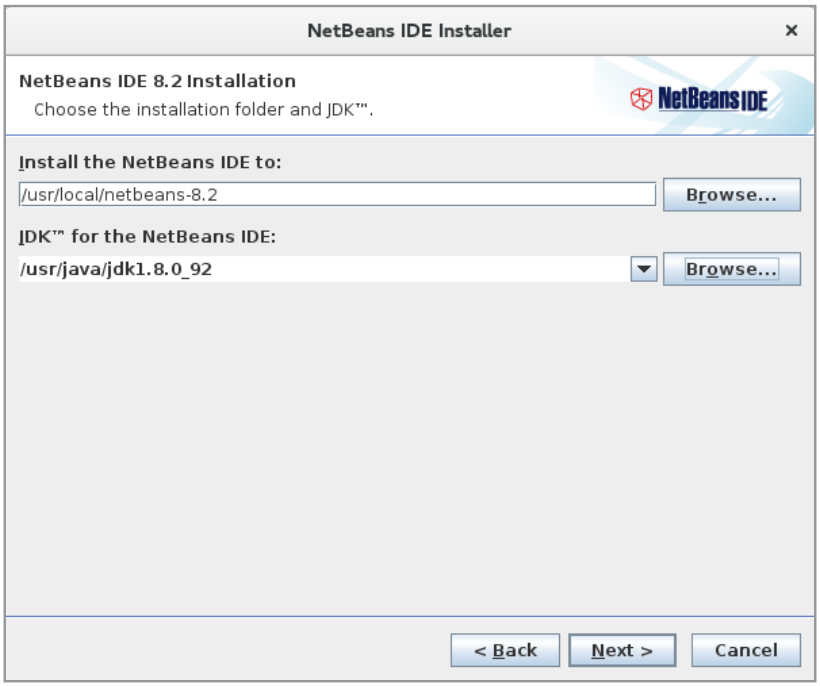
After the package is downloaded you must give execute right, after which the program is installed with the command (assuming that your current directory is where the package is downloaded):

```
sudo sh netbeans-8.2-javaee-linux.sh
```

Then the rest goes by itself (Fedora 23). However, you must specify where your Java is installed (see below). Once the installation is complete, both *NetBeans* and *Glassfish* are installed.

In some Fedora distributions, I have encountered problems (Fedora 25), where the installation program terminates with an exception. If you meet it, try the following commands as *root*:

```
xhost +
sh netbeans-8.2-javaee-linux.sh
xhost -
```



Back there is only the start of the Glassfish server. Open a terminal and set your current directory to the glassfish server, for example

```
cd /usr/local/glassfish-4.1.1/bin
```

After that, the server can be started with the command:

```
sudo sh asadmin start-domain domain1
```

You can test that the server is running by enter the address in the browser:

```
localhost:8080
```

and in the same way you can test that the admin server is running by entering the address

```
localhost:4848
```

If you want to stop the server, you can use the command

```
sudo sh asadmin stop-domain domain1
```

and if you want to install the server as a service, you can use the command:

```
sudo sh asadmin create-service domain1
```

APPENDIX B: HTTP

Before I go on, I will add a few remarks about the HTTP protocol. A web application, as already described, consists of a number of files (web pages) located on a web server. These files (web pages, images, etc.) are used by clients who download the files and display them in a browser. That is a client is a regular PC with a browser while a web server is a computer (a server) that runs a program (a service) called a web server. The web server constantly listens on a port (for example 8080) after requests from clients, and in case of a request (a request on a particular page), the web server sends a response as a HTML document to the address to which the client's request relates. Specifically, the client enters the address of a web page in his browser, for example.

www.torus.dk

and the browser will then convert the name to an IP address using the DNS system, after which the browser opens a socket to the server via a port (the default is 80). Then the web server can reply back with a response in the form of HTML, which the browser can then interpret and display as a web page.

From the server, more things can happen. In simplest cases a client request regarding a static HTML page, and if so, the server should do nothing but load the page from the machine's disk and send it to the client's browser. Today, however, it will often be a dynamic web page – for example a servlet and client's request regarding then in reality a program. The web server loads that application and executes it. Often it will include database operations either because the application requires data from a database or also because the client has sent data with the request, which must be stored in a database. In any case, the program that the web server executes will generate HTML (whose content may be dynamically based on the content of a database) and send this HTML to the client.

In order for this client/server architecture to work, agreements or rules that exactly determine the format of the client requests and server responses are required, and this is where HTTP enters the picture as the name of that protocol (a protocol is a set of rules), which indicates how the communication between client and server should be.

HTTP stands for *HyperText Transfer Protocol* and is a protocol belonging to the application layer, and typically implemented over TCP / IP. It is a stateless protocol that, as mentioned, is based on request and response, where a client application sends a request to a server and where the server responds to the client with a response. That the protocol is stateless means that after the server has sent the response, it has forgotten everything about the client. The server thus stores no information about its clients, and whenever a client requests the server, it is perceived by the server as a new request. That the HTTP protocol is thus stateless, presents many challenges in connection with web applications, and there are several techniques to cope with these challenges, where I have already mentioned both cookies and session objects. The current version of http is called version 1.1.

A URL is a reference to a particular resource (file) on the Internet and generally has the following format:

http://Servernavn/Filenavn

An example of an URL could be

```
http://www.eadania.dk/education/it/index.xhtml
```

Here *www.eadania.dk* is the server name, while *education/it/index.xhtml* is the name of the file on the server relative to a directory determined by the server. Exactly an URL format is

http://Servernavn:port/Filenavn

where port is the *port number* used by the web server, but if nothing is specified, the browser will use port 80, which is standard on the Internet.

In HTTP, a client request and response from a server consist of three parts:

- request- or response line
- a header
- data

A web-transaction consist of

- a client request
- a server response

and it is initiated by the client by establishing a connection to the server at a predetermined port. Default is, as mentioned, port 80. The client then sends an HTTP command followed by the document address and the version of the protocol that is used. It could, for example, be

GET /index.htm HTTP/1.1

Next, the header is sent, consisting of a number of lines of the form

Keyword: Value

where each line ends with a carriage return and a line feed. As an example, it could be

```
User-Agent:Lynx/2.4 libwww/5.1k
Accept:image/gif, image/x-xbitmap, image/jpeg, */*
```

After the last header line, a blank line is sent, after which it may follow data to be sent to the server and it all ends with another blank line.

A command associated with a request is also called for a method and there are 7 methods:

Method	Description
OPTIONS	Used to ask a server about what options it offers.
GET	Asks the server to return the document that the document address specifies.
HEAD	In principle, works as GET, but the server does not return the actual content of the document. It is used to test if the document has been changed since the last request.
POST	Used to send a data block to the server along with a request.
PUT	It is the opposite command to GET and stores the data block on the document address.
DELETE	Deletes the document on the server that the document address specifies.
TRACE	Used to track a request's route through different firewalls and proxy servers and used in connection with debugging of complex network issues.

As mentioned, a response from a server also consists of three parts. The first line has the format

Protocol Statuscode Description

and as an example it could be

```
HTTP/1.1 200 OK
```

Then, the server's header contains information about the server and the document that is being sent. The header consists of lines according to the same pattern as a request, and it ends in the same way with a blank line. After the header comes the data block, which may be a document, an output from a program or possibly an error message.

One of the differences between HTTP / 1.0 and HTTP / 1.1 is that in 1.1, the connection is not terminated after the server has completed its response. The reason is that many HTML documents contain references to other files such as images, etc., and the client may therefore request such resources without having to re-establish the connection.

The HTTP protocol defines many headers, but the most important is the following where Q stands for request while S is for response:

Header	Q	S	Description
Accept:	x		Specifies what types the client will accept.
Accept-Charset:	x		Specifies which character sets the browser can accept.
Accept-Encoding:	x		Specifies what types of encodings the client knows. If this header is omitted, the client will accept all encodings.
Accept-Language:	x		Specifies which languages the client accepts.
Age:		x	Used in conjunction with cache control.
Allow:		x	Specifies what methods the resource on the document address can respond to.
Authorization:	x		Used in conjunction with digital signatures.
Cache-Control:	x	x	Used by proxy servers and describes how to handle request and response.
Code:	x		Defines an encoding of the data block. Standard is Base64.
Content-Base:		x	Used to solve relative URLs in the returned document. This header overrides Content-Location.

Header	Q	S	Description
Content-Encodning:		x	Indicates an encoding applied to the document before it is transmitted.
Content-Language:		x	Indicates the natural language for the content.
Content-Length:		x	The length of the data block in bytes.
Content-Location:		x	Specifies the location of the document being sent.
Content-MD5:		x	Used for a checksum for the data block.
Content-Type:		x	Specifies the type of data being sent.
Expires:		x	Specifies a date for data to be perceived as obsolete.
From:	x		The client's email address.
Host:	x		The host's name from the URL.
Last-Modified:		x	Specifies where the document sent last has been changed.
Location:		x	Used for redirect to another address for example in case of an error.
Referrer:	x		The source for the current request.
User-Agent:	x		The browser's signature that is used to test which browser a request is coming from.
Warning:		x	Used for additional information in connection with a response.

As mentioned above, HTTP defines seven methods, and *GET* and *POST* are the most important. GET is the simplest and simply consists in sending a request to a server that it should send a document with a given document address. That is there is no data block in connection with a GET. The document that GET refers may be a program – for example a servlet – and you can transfer data to such a program. This happens as part of the URL:

```
GET /index.xhtml?code=7800&date=20030423 HTTP/1.1
```

and you would in the browser's address field types

```
http://server:8080/index.xhtml?code=7800&date=20030423
```

Here two data elements are transferred to the page *index.xphtml*. The question mark after the document address indicates that a parameter is sent in pairs

keyword=value

If more parameters are sent, each pair must be separated by &.

POST is different as data is encapsulated in the data block and the method is used when larger and more complex data sizes are sent to the server. With POST it is possible to enclose data of different types in the same request as well as to send binary data to the server. Another important difference between GET and POST is that when GET data is added to the URL, they are also visible to the user in the browser's address bar, which is not always desirable. This is not the case with the POST method. In addition, it should be noted that the server may have limitations regarding the length of an URL, and thus how many data can be sent with GET.