Spago Quick Start:
for Struts Developers

Author	Daniela Butano Gabriele Ruffatti

Index

Versions History

Version/Release n° :	1.0	**Data Version/Release :**	December, 9th 2004
Description:	First release		

Document Goal

This document walks you through the process of creating and configuring a sample application using Spago. I suppose that you know Struts: the goal is to aware you about the difference you will met when you will develop a new application using Spago.

In the chapter 3, I will introduce the mains features of Spago framework.

References

For more details about Spago see the follow documentation downloaded in http://spago.eng.it/docs_it/documentation/index.html:

- [1] *Spago Overview*
- [2] *Spago User Guide*
- [3] *Spago Tutorial*

1 The sample: Employee Application

In this chapter, I will walk you through developing all of the necessary components for a small application. The application you are going to create simulates entering an employee into a database. The user will be required to enter an employee's name and age.

1.1 INSTALL SOFTWARE

Download and install Tomcat 5.0.28. You can download Tomcat here: http://jakarta.apache.org/site/binindex.cgi.
Setting up Tomcat is not difficult but is out of the scope of this tutorial.

We will use the project template distributed in *http://sourceforge.net/project/showfiles.php?group_id=119426*, into the package Spago Samples in the startup-sample.zip. The template is released as *Spago.war*

1.2 THE APP DIRECTORY

Just a look at the structure of application directory in the {tomcat}/webapps/ folder (where {tomcat} equals the root directory of your Tomcat installation).

The directory structure should look like this:

STRUTS	SPAGO
webapps | demo_struts | --- WEB-INF | --- classes | --- lib ---commons-beanutils.jar ---commons-collections.jar ---commons-digester.jar --- commons-logging.jar ---jstl.jar ---standard.jar ---struts-el.jar ---struts.jar | --- src	webapps | demo_spago --- admin (Apache SOAP A.I.) --- css (Stylesheets used by the JSP) --- img (Images used by some tag libraries) --- jamon (JAMon performance monitor A.I.) --- jsp (Default error pages) --- WEB-INF | --- classes --- conf (SOAP configuration files) | --- spago (Spago config files) --- lib (Spago and needed external libraries) --- log (Directory for log files) --- tld (Tag libraries definitions) --- src

1.3 CREATE THE WEB.XML

First at all, you must create the *web.xml* file.

All requests will pass through a controller, since an MVC architecture is used.

Struts	**Spago**
The main controller of Struts is the *org.apache.struts.action.ActionServlet*. All your requests will pass to this servlet.	The main controller of Spago is the *it.eng.spago.dispatching.httpchannel.AdapterHTTP*. Spago is a framework multichannel. That is means every service can be called from different channels. All HTTP requests pass to this *AdapterHTTP*. All SOAP requests pass to *AdapterSOAP*. You must call the *AdpterEJB* adapter for the EJB channel.

The structure of the *web.xml* file of an application using Spago should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp">
        <display-name>Spago</display-name>
        <servlet>
                <servlet-name>ConfigServlet</servlet-name>
                <servlet-class>it.eng.spago.configuration.ConfigServlet</servlet-class>
                <!--init-param>
                        <param-name>AF_ROOT_PATH</param-name>
                        <param-value>C:\Progetti\spago\eclipse\Spago</param-value>
                </init-param-->
                <init-param>
                        <param-name>AF_CONFIG_FILE</param-name>
                        <param-value>\WEB-INF\conf\spago\master.xml</param-value>
                </init-param>
                <load-on-startup>0</load-on-startup>
        </servlet>
        <servlet>
                <servlet-name>AdapterHTTP</servlet-name>
                <servlet-class>it.eng.spago.dispatching.httpchannel.AdapterHTTP</servlet-
class>
        </servlet>
        <servlet-mapping>
                <servlet-name>AdapterHTTP</servlet-name>
                <url-pattern>/servlet/AdapterHTTP</url-pattern>
        </servlet-mapping>
        <session-config>
                <session-timeout>30</session-timeout>
        </session-config>
        <welcome-file-list>
                <welcome-file>/html/index.html</welcome-file>
        </welcome-file-list>
        <taglib>
                <taglib-uri>spagotags</taglib-uri>
                <taglib-location>/WEB-INF/tld/spago.tld</taglib-location>
        </taglib>
</web-app>
```

1.4 FROM STRUTS-CONFIG.XML TO SPAGO CONFIGURATION FILES

Struts	Spago
The controller ActionServlet will use this struts-config file to determine all relevant operations. This *struts-config.xml* file is the 'road map' of every application. This file will tell our requests where to go (usually an Action class), what Form beans to use, and where to go after the request submits.	Spago uses a lot of configuration files. The main file is the *master.xml,* having the references to all others. The *master.xml* path is defined in the *web.xml* file.

In this application, the *master.xml* structure should look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<MASTER data_access_configuration_file_path="/WEB-INF/conf/spago/data_access.xml">
	<CONFIGURATOR path="/WEB-INF/conf/spago/actions.xml" />
	<CONFIGURATOR path="/WEB-INF/conf/spago/common.xml" />
	<CONFIGURATOR path="/WEB-INF/conf/spago/dispatchers.xml" />
	<CONFIGURATOR path="/WEB-INF/conf/spago/initializers.xml" />
	<CONFIGURATOR path="/WEB-INF/conf/spago/modules.xml" />
	<CONFIGURATOR path="/WEB-INF/conf/spago/pages.xml" />
	<CONFIGURATOR path="/WEB-INF/conf/spago/presentation.xml" />
	<CONFIGURATOR path="/WEB-INF/conf/spago/publishers.xml" />
	<CONFIGURATOR path="/WEB-INF/conf/spago/statements.xml" />
	<CONFIGURATOR path="/WEB-INF/conf/spago/tracing.xml" />
</MASTER>
```

Struts | **Spago**

```
<struts-config>
    <!-- Form Bean Definitions -->
    <form-beans>
        <form-bean            name="employeeForm"
type="demo.EmployeeForm"/>
    </form-beans>
```

Spago doesn’t use an ActionForm holding the data submitted from JSP.

```
<!-- Global forwards -->
    <global-forwards>
        <forward name="error" path="/error.jsp"/>
    </global-forwards>
```

In Spago, every service (implemented by action mode or modules mode) has an own publisher, declared in the *presentation.xml* file.

```
<!--  Action Mapping Definitions  -->
    <action-mappings>
            <action path="/setupEmployeeForm"
            type="demo.SetUpEmployeeAction"
            name="employeeForm"
            scope="request"
            validate="false"
            >
            <forward
                    name="continue"
                    path="/employeeForm.jsp"/>
        </action>

        <action path="/insertEmployee"
            type="demo.InsertEmployeeAction"
            name="employeeForm"
```

In Spago, the similar settings are in the files: *action.xml, presentation.xml, publishers.xml*.

```
            scope="request"
            validate="true"
            input="/employeeForm.jsp"
            >
            <forward
                    name="success"
                    path="/confirmation.jsp"/>
        </action>

    </action-mappings>
```

In Spago, the actions are defined in the *actions.xml* file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ACTIONS>
      <ACTION                                                       name="publishAction"
class="it.eng.spago.dispatching.action.util.PublishAction" scope="REQUEST">
            <CONFIG></CONFIG>
      </ACTION>
      <ACTION       name="setupEmployeeForm"       class="demo.action.SetupEmployeeAction"
scope="REQUEST">
            <CONFIG></CONFIG>
      </ACTION>
      <ACTION        name="insertEmployee"        class="demo.action.InsertEmployeeAction"
scope="REQUEST">
            <CONFIG></CONFIG>
      </ACTION>
</ACTIONS>
```

The association between the actions and the publishers is defined in the *presentation.xml* file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PRESENTATION>
      <MAPPING            business_type="ACTION"            business_name="publishAction"
publisher_name="genericPublisher" />
      <MAPPING          business_type="ACTION"          business_name="setupEmployeeForm"
publisher_name="setupPublisher" />
      <MAPPING           business_type="ACTION"           business_name="insertEmployee"
publisher_name="genericPublisher" />
</PRESENTATION>
```

the publishers are defined in the *publishers.xml* file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PUBLISHERS>
      <PUBLISHER name="SECURITY_ERROR_PUBLISHER">
            <RENDERING channel="HTTP" type="JSP" mode="FORWARD">
                  <RESOURCES>
                        <ITEM prog="0" resource="/jsp/spago/securityError.jsp" />
                  </RESOURCES>
            </RENDERING>
      </PUBLISHER>
      <PUBLISHER name="SERVICE_ERROR_PUBLISHER">
            <RENDERING channel="HTTP" type="JSP" mode="FORWARD">
                  <RESOURCES>
                        <ITEM prog="0" resource="/jsp/spago/serviceError.jsp" />
                  </RESOURCES>
            </RENDERING>
      </PUBLISHER>
      <PUBLISHER name="setupPublisher">
            <RENDERING channel="HTTP" type="JSP" mode="FORWARD">
                  <RESOURCES>
```

```
                <ITEM prog="0" resource="/jsp/demo/employeeForm.jsp" />

            </RESOURCES>
        </RENDERING>
    </PUBLISHER>
    <PUBLISHER name="genericPublisher">
        <RENDERING channel="HTTP" type="JAVA" mode="">
            <RESOURCES>
            <ITEM                                                   prog="0"
resource="it.eng.spago.presentation.DefaultPublisherDispatcher">
            <CONFIG>
            <CHECKS>
                <CHECK target="menuPublisher">
                <CONDITIONS>
                <PARAMETER name="PUBLISHER" scope="SERVICE_REQUEST" value="MENU" />
                </CONDITIONS>
                </CHECK>
                <CHECK target="confirmPublisher">
                <CONDITIONS>
                <PARAMETER name="1" scope="ERROR" value="AF_NOT_DEFINED" />
                <PARAMETER         name="PUBLISHER"          scope="SERVICE_REQUEST"
value="AF_NOT_DEFINED" />
                </CONDITIONS>
                </CHECK>
                <CHECK target="setupPublisher">
                <CONDITIONS>
                <PARAMETER name="1" scope="ERROR" value="AF_DEFINED" />
                </CONDITIONS>
                </CHECK>
            </CHECKS>
            </CONFIG>
            </ITEM>
            </RESOURCES>
        </RENDERING>
    </PUBLISHER>
    <PUBLISHER name="menuPublisher">
        <RENDERING channel="HTTP" type="JSP" mode="FORWARD">
            <RESOURCES>
                <ITEM prog="0" resource="/jsp/demo/menu.jsp" />
            </RESOURCES>
        </RENDERING>
    </PUBLISHER>
    <PUBLISHER name="confirmPublisher">
        <RENDERING channel="HTTP" type="JSP" mode="FORWARD">
            <RESOURCES>
                <ITEM prog="0" resource="/jsp/demo/confirmation.jsp" />
            </RESOURCES>
        </RENDERING>
    </PUBLISHER>
</PUBLISHERS>
```

The publisher associates more <RENDERING> to different channels: HTTP, WAP, SOAP, EJB
The publisher's type can be: JSP, XSL, JAVA, LOOP. In this sample, the `genericPublisher` is a JAVA publisher configurable in the section `CONFIG`. For more details see the *Spago User Guide*.

Struts	Spago

`<!-- message resources -->`
`<message-resources`
`parameter="ApplicationResources"`
`null="false" />`
`</struts-config>`
	Spago uses the properties file *message_languagecode_countrycode.properties* to hold display items for the JSPs and all messages

1.5 ACTION FORM

Struts	Spago
Struts uses the ActionForm objects to hold the submitted JSP form data. (They could also be used for displaying results back on a page, although this is a less common function). The ActionForm should contain String fields representing the properties in the EmployeeDTO	In Spago all data submitted are hold in the *SourceBean* object. The *SourceBean* is an objects container; a key identifies every object. The *SourceBean* can be displayed as XML structure, so the business logic doesn't depend on channel. For more details see the *Spago User Guide*.

1.6 CREATE THE DATA TRANSFER OBJECT

Since you are dealing with an Employee to insert, you need a way to store information about this Employee that you could hand off to a business object (the model layer of MVC). The model layer will be responsible for doing the actual insert. So you need a class representing the employee. You will create a bean that has just a couple of fields and appropriate get and set methods. Since this object will transfer stored information from one part of our application to another it is called a Data Transfer Object (or often Value Object). Create the EmployeeDTO:

```
package demo.dto;
public class EmployeeDTO {
    private String name;
    private int age;
    private String department;

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

    public void setAge(int age) {
        this.age = age;
    }

    public void setDepartment(String department) {
        this.department = department;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String getDepartment() {
        return department;
    }
}
```

1.7 CREATE THE BUSINESS SERVICE

You need to create at least one object that will act as a transition object to the model layer of our MVC application.

You will create an EmployeeService class to handle the small amount of business logic we have. Create the EmployeeService:

```
package demo.service;

import demo.dao.EmployeeDAO;
import demo.dto.EmployeeDTO;

public class EmployeeService {
   public boolean insertEmployee(EmployeeDTO employee ) {
        return EmployeeDAO.insert(employee);
   }
}
```

1.8 CREATE DAO

You create an EmployeeDAO to execute the sqlcommand. In the *statements.xml* file you add the statement `INSERT_EMPLOYEE:`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<STATEMENTS>
     <STATEMENT name="INSERT_EMPLOYEE" query="INSERT INTO EMPLOYEE VALUES(?,?,?)"/>
</STATEMENTS>
```

In *data_access.xml* we have to configure the connection pool (See the *Spago User Guide* for more details) an then create the EmployeeDAO:

```
package demo.dao;

import it.eng.spago.dbaccess.DataConnectionManager;
import it.eng.spago.dbaccess.SQLStatements;
import it.eng.spago.dbaccess.Utils;
import it.eng.spago.dbaccess.sql.DataConnection;
import it.eng.spago.dbaccess.sql.SQLCommand;
import it.eng.spago.dbaccess.sql.result.DataResult;
import it.eng.spago.dbaccess.sql.result.InformationDataResult;
import it.eng.spago.tracing.TracerSingleton;

import java.sql.Types;
import java.util.ArrayList;

import demo.dto.EmployeeDTO;

public class EmployeeDAO {
     public static boolean insert(EmployeeDTO employeeDTO) {
          Object result = null;
         DataConnectionManager dataConnectionManager = null;
         DataConnection dataConnection = null;
         SQLCommand sqlCommand = null;
         DataResult dataResult = null;
         ArrayList inputParameters = new ArrayList();
         InformationDataResult informationDataResult = null;
```

```
        try {
          dataConnectionManager = DataConnectionManager.getInstance();
          dataConnection = dataConnectionManager.getConnection("employeedemo");
             inputParameters.add(dataConnection.createDataField("",        Types.VARCHAR,
employeeDTO.getName()));
             inputParameters.add(dataConnection.createDataField("",   Types.INTEGER,   new
Integer(employeeDTO.getAge())));
             inputParameters.add(dataConnection.createDataField("",        Types.VARCHAR,
employeeDTO.getDepartment()));
          String statement = SQLStatements.getStatement("INSERT_EMPLOYEE");
            sqlCommand = dataConnection.createInsertCommand(statement);
            dataResult = sqlCommand.execute(inputParameters);
            informationDataResult = (InformationDataResult) dataResult.getDataObject();
         } catch (Exception ex) {
                TracerSingleton.log("Demo",                        TracerSingleton.CRITICAL,
"EmployeeDAO::insert:", ex);
                return false;
        }
        finally {
          Utils.releaseResources(dataConnection, sqlCommand, dataResult);
        }

        if (informationDataResult.getAffectedRows() != 1)
          return false;
        return true;
        }
}
```

1.9 CREATE THE ACTION CLASSES

You often might want to pre-populate some information the JSP page might need. One way you can do this is by submitting to an Action class before forwarding to your JSP page. Instead of forward directly to the employeeForm.jsp, you are going to first submit to a SetupEmployeeAction which code (we see only the code for Spago) should look like this:

```
package demo.action;

import it.eng.spago.base.SourceBean;
import it.eng.spago.dispatching.action.AbstractAction;
import it.eng.spago.tracing.TracerSingleton;

public class SetupEmployeeAction extends AbstractAction {

     public SetupEmployeeAction () {
     }

     public void service(SourceBean request, SourceBean response) throws Exception {
          TracerSingleton.log("Demo",                        TracerSingleton.INFORMATION,
"SetupEmployeeAction::service: invocato");
          response.setAttribute("department", "department1");
     }
}
```

A few notes:

- The action extends the `AbstractAction` class, in this way is possible to access the *RequestConstainer* and then *SessionContainer* throws *getRequestContainer()* and *getRequestContainer().getSessionContainer()* methods.
- The *RequestContainer* contains some information dependent on channel: the HTTP request, the ServletConfig and the *servicerequest*,. The *servicerequest* is the `request`, input parameter for all actions. The `request` holds all parameters from *HTTPRequest*
- The action has as input parameter the `request` and the `response`. These objects are the *SourceBean*, so the business logic is independent from channel.

Now, we have to create the InsertEmployeeAction responsible for calling our EmployeeService class and passing it the EmployeeDTO to perform the insert.

```
package demo.action;

import demo.dto.EmployeeDTO;
import demo.service.EmployeeService;
import it.eng.spago.base.SourceBean;
import it.eng.spago.base.SourceBeanException;
import it.eng.spago.dispatching.action.AbstractAction;
import it.eng.spago.error.EMFErrorSeverity;
import it.eng.spago.error.EMFUserError;
import it.eng.spago.tracing.TracerSingleton;

public class InsertEmployeeAction extends AbstractAction {

     public InsertEmployeeAction () {
     }

     public void service(SourceBean request, SourceBean response) throws Exception {
          TracerSingleton.log("Demo",                     TracerSingleton.INFORMATION,
"InsertEmployeeAction::service: invocato");
          EmployeeDTO employeeDTO = new EmployeeDTO();
          employeeDTO.setName((String)request.getAttribute("name"));
          employeeDTO.setAge(Integer.parseInt((String)request.getAttribute("age")));
          employeeDTO.setDepartment((String)request.getAttribute("department"));
          EmployeeService service = new EmployeeService();
         if (!service.insertEmployee(employeeDTO)) {
               TracerSingleton.log("Demo",                     TracerSingleton.MAJOR,
"InsertEmployeeAction::service: inserimento non effettuato");
               getErrorHandler().addError(new
EMFUserError(EMFErrorSeverity.BLOCKING,1));
         }

          response.setAttribute("name", employeeDTO.getName());
          response.setAttribute("age", Integer.toString(employeeDTO.getAge()));
          response.setAttribute("department", employeeDTO.getDepartment());

     }
}
```

A few notes:

- In the service method, it is possible to access the error handler (`getErrorHandler()`) and add a new user error which message is defined in the file properties as:

  `1 = Error during insert`

1.10 CREATE INDEX.HTML

Create the index.html page. This is the first page called.

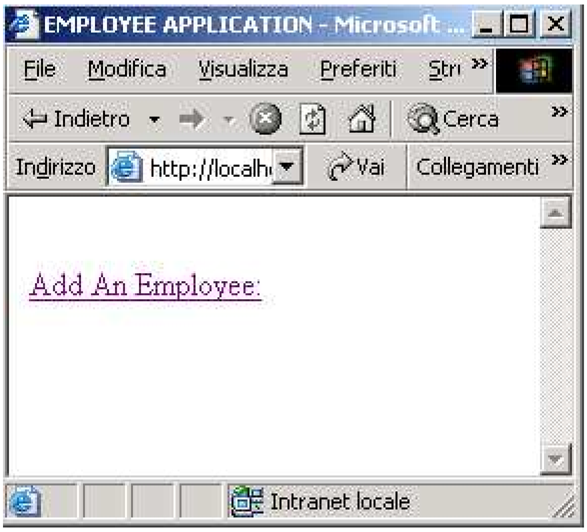
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<html>
<head>
       <title></title>
</head>
<body
onload="window.location='servlet/AdapterHTTP?ACTION_NAME=publishAction&PUBLISHER=MENU&NEW
_SESSION=TRUE'"/>
</html>
```

A few notes:

- We use the `ACTION_NAME` attribute to call the action defined in the *actions.xml* file .
- The `publishAction` action released in Spago permits publishing the first jsp page dispayed *menu.jsp*
- To start the application you need the `NEW_SESSION=TRUE` parameter; Spago throws an *SessionExpiredException* if you don't use it.

1.11 CREATE MENU.JSP

Create the menu.jsp page. This is the first page displayed:



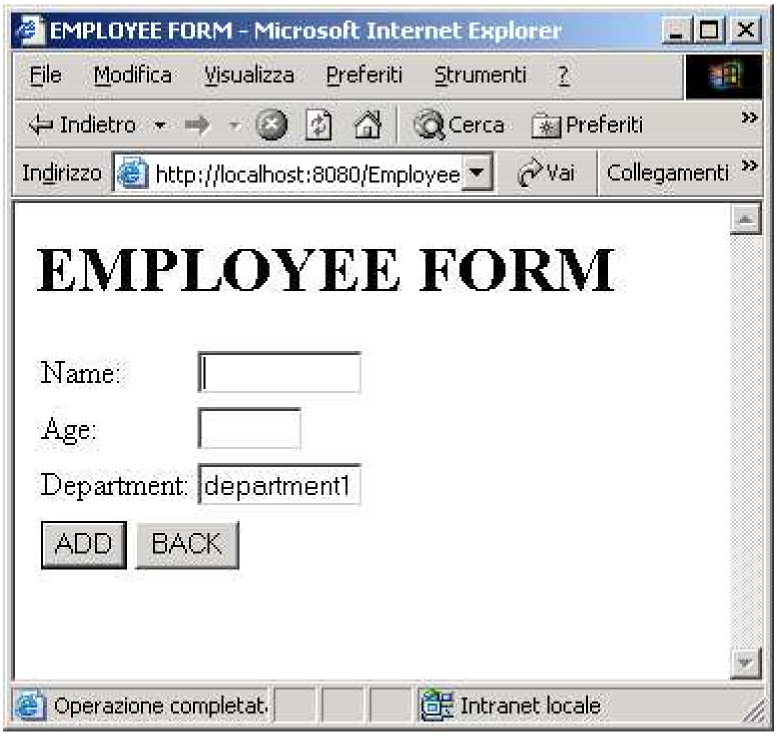
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<%@ taglib uri="spagotags" prefix="spago" %>
<%@ page
language="java"
extends="it.eng.spago.dispatching.httpchannel.AbstractHttpJspPage"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
session="true"
errorPage="/jsp/spago/jspError.jsp"
%>
<html>
<head>
      <title><spago:message code="title.employeeApp"/></title>
</head>
<body>
      <br>
      <a                    href="AdapterHTTP?ACTION_NAME=setupEmployeeForm"><spago:message
code="href.employeeadd"/></a>
</body>
</html>
```

A few notes:

- Page extends "`it.eng.spago.dispatching.httpchannel.AbstractHttpJspPage`": in this way you can access the response container, the response service, the error handler, etc.
- The statement `<%@ taglib uri="spagotags" prefix="spago"%>` allows you to use the Spago tag library. In this JSP, we use the tag *message* to display the messages holding in the *message_languagecode_countrycode.properties* file matching with attribute `code`
- The form's submit calls the insert action, as you can see from the attribute `action="../servlet/AdapterHTTP?ACTION_NAME=insertEmployee"`.

1.12 CREATE EMPLOYEEFORM.JSP

Create the employeeform.jsp in this way.



http://localhost:8080/Employee

EMPLOYEE FORM

Name:

Age:

Department: department1

ADD BACK

```
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<%@ taglib uri="spagotags" prefix="spago" %>
<%@ page
language="java"
extends="it.eng.spago.dispatching.httpchannel.AbstractHttpJspPage"
import="it.eng.spago.base.SourceBean"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
session="true"
errorPage="/jsp/spago/jspError.jsp"
%>
<title><spago:message code="title.employee.employeeform"/></title>
</head>
<body>
     <h1><spago:message code="title.employee.employeeform"/></h1>
     <html>
     <BODY onload="checkError();">
     <spago:error />
     <form name="employeeForm" action="AdapterHTTP?ACTION_NAME=insertEmployee"
method="post">
     <%SourceBean serviceResponse = getServiceResponse(request);%>
          <table>
          <tr>
               <td><spago:message code="label.name"/></td>
               <td><input type="text" size="10" name="name"
value="<%=(String)serviceResponse.getAttribute("name")%>"></td>
          </tr>
          <tr>
               <td><spago:message code="label.age"/></td>
               <td><input type="text" size="5" name="age" value=""></td>
          </tr>
          <tr>
               <td><spago:message code="label.department"/></td>
```

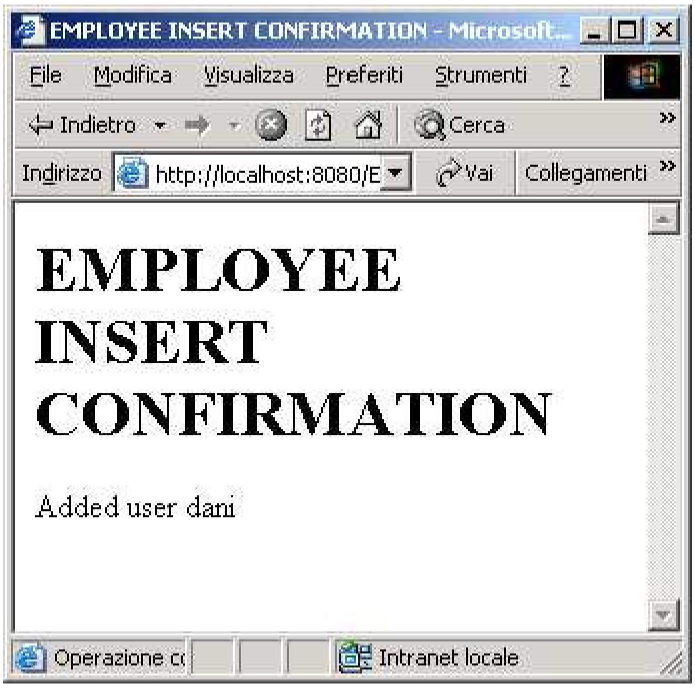
```
                <td><input        type="text"        size="10"        name="department"
value="<%=(String)serviceResponse.getAttribute("department")%>"</td>
            </tr>
            </table>
            <table>
            <tr>
                <td><INPUT      TYPE="submit"      name="save"      value="<spago:message
code='buttons.submit'/>"></td>
                <td><INPUT      TYPE=BUTTON      name="back"      value="<spago:message
code='buttons.back'/>" onClick="window.location='AdapterHTTP?NAVIGATOR_BACK=1'"></td>
            </tr>
            </table>
        </form>
</body>
</html>
```

A few notes:

- Page extends "`it.eng.spago.dispatching.httpchannel.`**`AbstractHttpJspPage`**": so you can access the response container, the response service, the error handler, etc.
- The statement `<%@ taglib uri="spagotags" prefix="spago"%>` allows you to use the Spago tag library. In this JSP, we use the tag *error* to display the errors holding in the error handler. The tag error builds the function javascript `checkError` that shows the error's stack using a pop-up. You have to implement a new tag for a different display.
- The form's submit calls the insert action, as you can see from the attribute `action="../servlet/AdapterHTTP?ACTION_NAME=insertEmployee".`
- The department's value is retrieved from the *responseService* through the method `getServiceResponse(request).`
- Spago implements a specific service called *Navigator*, which simplify the navigation. The *Navigator* maintains a request's stack of expired services and allows to re-execute a previous service in the same Request Container and Session Container conditions. For instance, it allows to handle a "browser refresh" or "browser back" in a more complex way. The command `NAVIGATOR_RESET=1` submit the last request and too the session state goes back. For more details, see the *Spago User Guide*.
- If the JSP page generates an exception, the `jsp/spago/error.jsp` will be displayed.

1.13 CREATE CONFIRMATION.JSP

Create the *confirmation.jsp* page in this way.

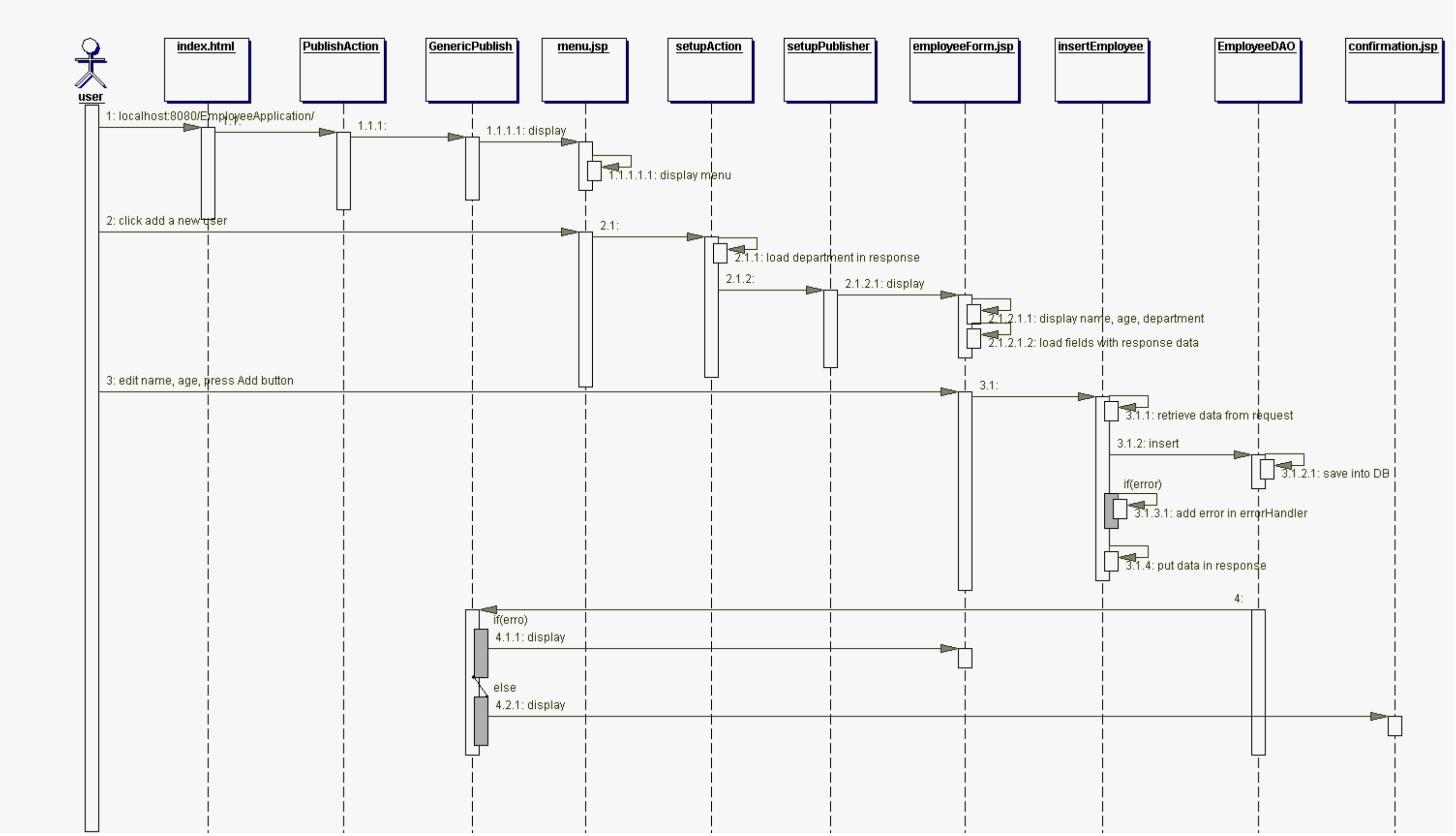


EMPLOYEE INSERT CONFIRMATION

Added user dani

```
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<%@ taglib uri="spagotags" prefix="spago" %>
<%@ page
language="java"
extends="it.eng.spago.dispatching.httpchannel.AbstractHttpJspPage"
import="it.eng.spago.base.SourceBean"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
session="true"
errorPage="/jsp/spago/jspError.jsp"
%>
<title><spago:message code="title.employee.insert.confirmation"/></title>
</head>
<body>
     <h1><spago:message code="title.employee.insert.confirmation"/></h1>
     <html>
     <%SourceBean serviceResponse = getServiceResponse(request);%>
     Added user <%=(String)serviceResponse.getAttribute("name")%>
</body>
</html>
```

1.14 EMPLOYEE APPLICATION SEQUENCE DIAGRAM



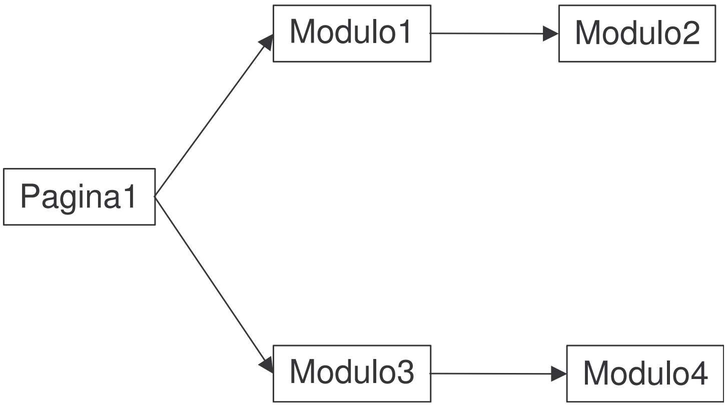
2 Spago features

In this chapter, a brief introduction to mains feature of Spago project not used for the sample.

2.1 MODULES DISPATCHING

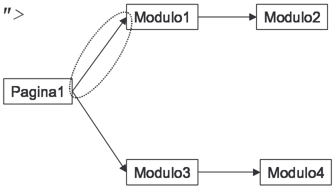
In the employee sample we use the **action mode**; in this mode the actions are the business objects which totally carry-out a request of an applicative service; one service corresponds to one business object (but the same object can carry out different requests).

Spago implements a **module mode** too; every module is a business object; it can cooperate with other modules to carry out a service request. All modules cooperating for the same service comprise a logical unit called a **page**. The service response is the union of all modules responses. An easy business logic execution workflow describes the order and conditions of execution for all the modules of one page. At the end of execution of each module, the framework identifies the next modules to execute according to request parameters. In the follow example, we implement a workflow, where the order of execution is: Module1, Module2, Module3, Module4.

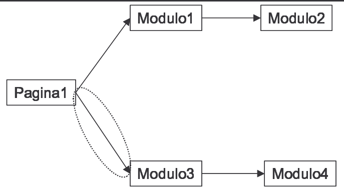


We define the *pages.xml* file with the follow structure:

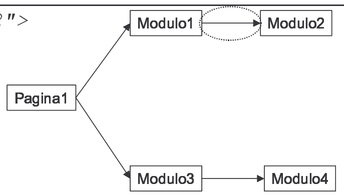
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PAGES>
    <PAGE name="Pagina1" scope="SESSION">
        <MODULES>
            <MODULE name="Modulo1"/>
            <MODULE name="Modulo2"/>
            <MODULE name="Modulo3"/>
            <MODULE name="Modulo4"/>
        </MODULES>
        <DEPENDENCIES>
            <DEPENDENCE source="Pagina1" target="Modulo1">
                <CONDITIONS>
                </CONDITIONS>
                <CONSEQUENCES/>
            </DEPENDENCE>
```



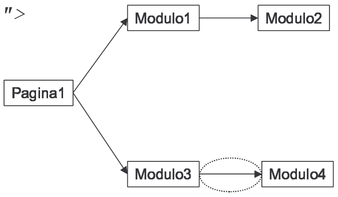
```
      <DEPENDENCE source="Pagina1" target="Modulo3">
                <CONDITIONS>
                </CONDITIONS>
                <CONSEQUENCES/>
            </DEPENDENCE>
```



```
            <DEPENDENCE source="Modulo1" target="Modulo2">
                <CONDITIONS>
                </CONDITIONS>
                <CONSEQUENCES/>
            </DEPENDENCE>
```



```
            <DEPENDENCE source="Modulo3" target="Modulo4">
                <CONDITIONS>
                </CONDITIONS>
                <CONSEQUENCES/>
            </DEPENDENCE>
```



```
        </DEPENDENCIES>
    </PAGE>
</PAGES>
```

The **conditions** are the rules to verify that a transition from one module to another exists; the **consequences** are the rules to define request parameters of one module. **Dependence** is the relation that verifies a set of conditions, , at the end of the execution of a source module and invokes a receiver module using some parameters corresponding to a consequence set.

If you carry out business logic in this way you have to acquire a lot of data. Otherwise, you can assign to every object specific responsibilities and you can **reuse** it for different services.

The module are defined in the *modules.xml* file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<MODULES>
                   <MODULE name="Modulo1" class="demo.Modulo1"/>
                  <MODULE name="Modulo2" class="demo.Modulo2"/>
                   <MODULE name="Modulo3" class="demo.Modulo3"/>
                   <MODULE name="Modulo4" class="demo.Modulo4"/>
</MODULES>
```

For more detail see the *Spago User Guide*.

2.1.1 VALIDATION

You can validate form-data server side using a specific component. It supplies validation for standard data types and it's extensible with custom java validation classes.

For details see the *Spago User Guide*.

2.2 LISTS AND DETAILS

Spago provides modules and tag libraries to generate lists and details configurable by XML files. It is possible to extend the functionality with personalized tags.
The lists use a pagination to handler data from database.
Spago implement two pagination mechanism:
One-shot: all data retrieved once from database. In this way you can have results waiting problems.
By *cache*: the data are retrieved in a different time, you can set the number of rows getting

You can download the samples of users list and detail, implemented by action mode or module mode at *http://sourceforge.net/project/showfiles.php?group_id=119426*, in the package Spago Samples (misc-samples.zip).

See the *Spago User Guide* for more details.