DbForms – a toolkit for rapidly building web-based database applications

by Joachim Peer

Many developers find themselves writing similar JSP and Servlet - code again and again when creating web-based database applications. The Open Source project DbForms (http://www.dbforms.org) provides a solution which reduces the amount of coding to an absolute minimum.

DbForms is a Java - based Rapid Application Development (RAD) - environment which enables developers to build web based database applications in very short time and under very little efforts.

This article gives an overview of DbForms, its concepts and features, and shows some code examples.

---

1 Introduction

DbForms enables developers to build web-based database driven applications in very short time and under very little efforts. DbForms – applications are built in a conceptually very similar manner to RAD - database building tools[1] like Microsoft Access (for Windows-based applications) or Sybase PowerSite (for web-based applications): The basic principle of these RAD-Tools could be described shortly as "placing database-aware components and action elements on templates (forms) which get executed at runtime."

DbForms builds on top of Java Servlets 2.2 and Java Server Pages 1.1 technology by Sun Microsystems. It makes extensive use of the JSP Tag Library extension introduced in the JSP 1.1 specification.

The project's homepage is located at **http://www.dbforms.org** and contains a complete user manual, several technical articles, source and binary distributions, online examples, a CVS, mailing lists and lots of other information related to DbForms.

2 The MVC-paradigma, incorporated by DbForms

DbForms implements the concepts of the Model-View-Controller (MVC) design pattern [Gamma].

The developer does **not** need to provide any *Controller*-related code. The developer just focuses on defining the *Model* and developing the JSP - *View* components of the application (mainly using DbForms custom tags).

The following paragraphs discuss how Model, View and Controller interact in DbForms.

[1] All named products are properties of their respective owners.

2.1 The Model: database objects described by database metadata

The use of DbForms is to perform operations on databases. The database-tables and -views of a DbForms-application must be declared in a XML – configuration file (usually stored as `WEB-INF/dbforms-config.xml`), which will be parsed and evaluated at Web-Application startup time.

```
<dbforms-config xmlns="http://www.wap-force.net/dbforms">

  <table name="customer">
    <field name="id" fieldType="int" isKey="true" />
    <field name="firstname" fieldType="char" />
    <field name="lastname" fieldType="char" />
    <field name="address" fieldType="char" />
  </table>

  <table name="orders">
    <field name="orderid" fieldType="int" isKey="true" />
    <field name="customerid" fieldType="int" isKey="true" autoInc="true" />
    <field name="date" fieldType="char" />
    <field name="annotation" fieldType="char" />
    <field name="amount" fieldType="int" />
  </table>

  <dbconnection
     name = "jdbc/dbformstest"
     isJndi = "true"
  />

</dbforms-config>
```

Listing 1 - defining the model

As shown in Listing 1, every table or view to be accessed by DbForms has to be declared inside a `<table>` element. All relevant table fields need to be declared inside a `<field>` element nested within their respective table-element.

There exists a **tool** for generating this XML-data **automatically**. The tool reads database metadata of a specified database and constructs a configuration file as shown in Listing 1.

2.2 The Controller: parsing, dispatching and executing events

As pointed out above, the developer does **not** need to provide any *Controller*-related code. The Controller is a true infrastructural component which should be transparent to the developer. However, it's useful if the developer has an idea of what is going on in the Controller. The following lines provide that information.

The Controller includes several components:

- **Controller-Servlet**: this servlet is the single-point-of-entry for all incoming HTTP-requests form clients.

- **EventEngine**: an assistant to the Controller-servlet - it focuses on filtering requests for WebEvents and instantiates them

- **WebEvent**-Objects: all Objects derived from the abstract super-class `WebEvent` have the ability to initialise themselves by reading a given HTTP-request. These events get executed either by the Controller directly or by the View.

The following description of the execution of a typical user-action should give you a better picture of what the controller does and how it interacts with the other components:

1. User presses the button "delete row" on his/her DbForms – application.

2. Client browser submits data via a HTTP-POST to the Controller-servlet

3. The Controller-servlet delegates the incoming request to the EventEngine which determinates the main-event (the event, the user *explicitly* triggered by clicking a button) However, there may occur *implicit* Events, too – i.e. automatic updating of all changed input fields of all data rows

4. The EventEngine-component parses that request and determinates the kind of action the user wants to be executed.

5. It then creates the appropriate WebEvent (in our case: a DeleteEvent) and delegates the Request-Object to this new created WebEvent which finalises its own initialisation. After that, the EventEngine returns the recently created and initialised event back to the Controller.

6. The Controller tells the event to execute its built-in operation, if it is a Database-Event. Other events (Navigation-Events, .etc.) are delegated to the appropriate View-component.

7. After that, the controller invokes EventEngine again to check if there are additional (implicit) events to be executed. If so, the appropriated WebEvents-Objects are created and executed in the same manner as the main event described above.

8. After that, the controller determinates the View-component the request should be forwarded to. If found, the controller invokes it and forwards the request.

9. If the view-component is a JSP-page containing DbForms – tags, those tags will search for navigation- events to be executed and after that they will finally generate the response for the user.

10. The response is rendered by the user's web browser.

2.3 The View: JSP templates provided by the Application developer

The view-portion of a DbForms- Application is generally constructed using **JSP-technology**. JSP-files may contain static HTML-Elements as well as dynamic elements containing Java-Code (definitions, statements, expressions). For more information about JSP please look into [JSP]

With release 1.1 of the JSP-API a powerful facility called "**JSP tag libraries**" [Taglib] was added. With these custom tags you can encapsulate even most sophisticated Java-code into lightweight JSP-tags.

DbForms makes use of the great potential of JSP tag libraries. It contains an extensive custom tags library for placing rendering and manipulating database data.

2.3.1 The structure of a DBForms - view

Figure 1 gives a conceptual overview of the main components of a typical DbForms-View:

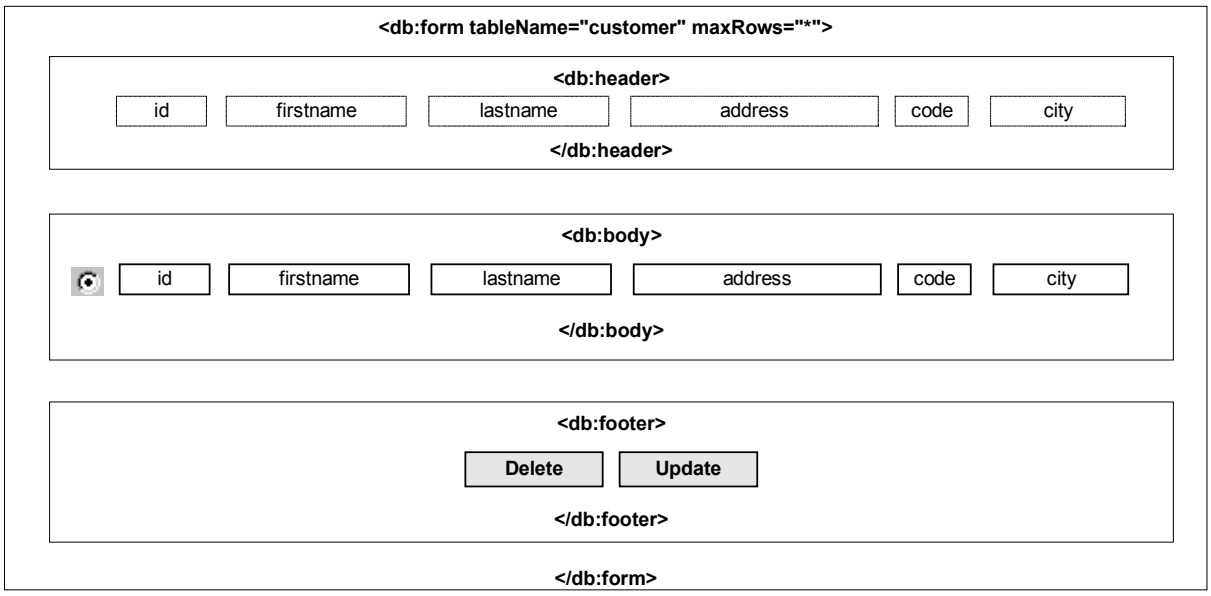

Figure 1- overall structure of a simple DbForms – View

2.3.2 The basis concepts of "form"s

Each DbForms-View-JSP may have one or more root tags of the type "**form**". Every "form"-tag has to contain exactly one "header" – tag, exactly one "body" – tag and exactly one "footer"-tag, in exactly that order.

Each of those tags may contain sub-elements like Data-Fields, Input-Fields, Action-Buttons, and - of course - plain HTML text and JSP code.

"Header" and "footer"- tags are commonly used for titles of pages, for labelling tables, for placing action and navigation- buttons, input-fields to enter new data, etc.

"Header" and "footer"- tags get evaluated only once.

The "body" tag is used for showing data-rows coming from the database, and for providing the user with functionality to edit that data. How many times the body tag and its sub-

elements get rendered depends on the value of the "**maxRows**" attribute of the form – element (and of course, on the number of datasets actually fetched from the database)

- maxRows = n => body gets executed n times at maximum (with n ∈ N)
- maxRows = "*" => body gets executed for every row in the table (=> "endless" form)

Nested forms

Every form may contain (one or more) nested sub-forms inside its "body" – element.

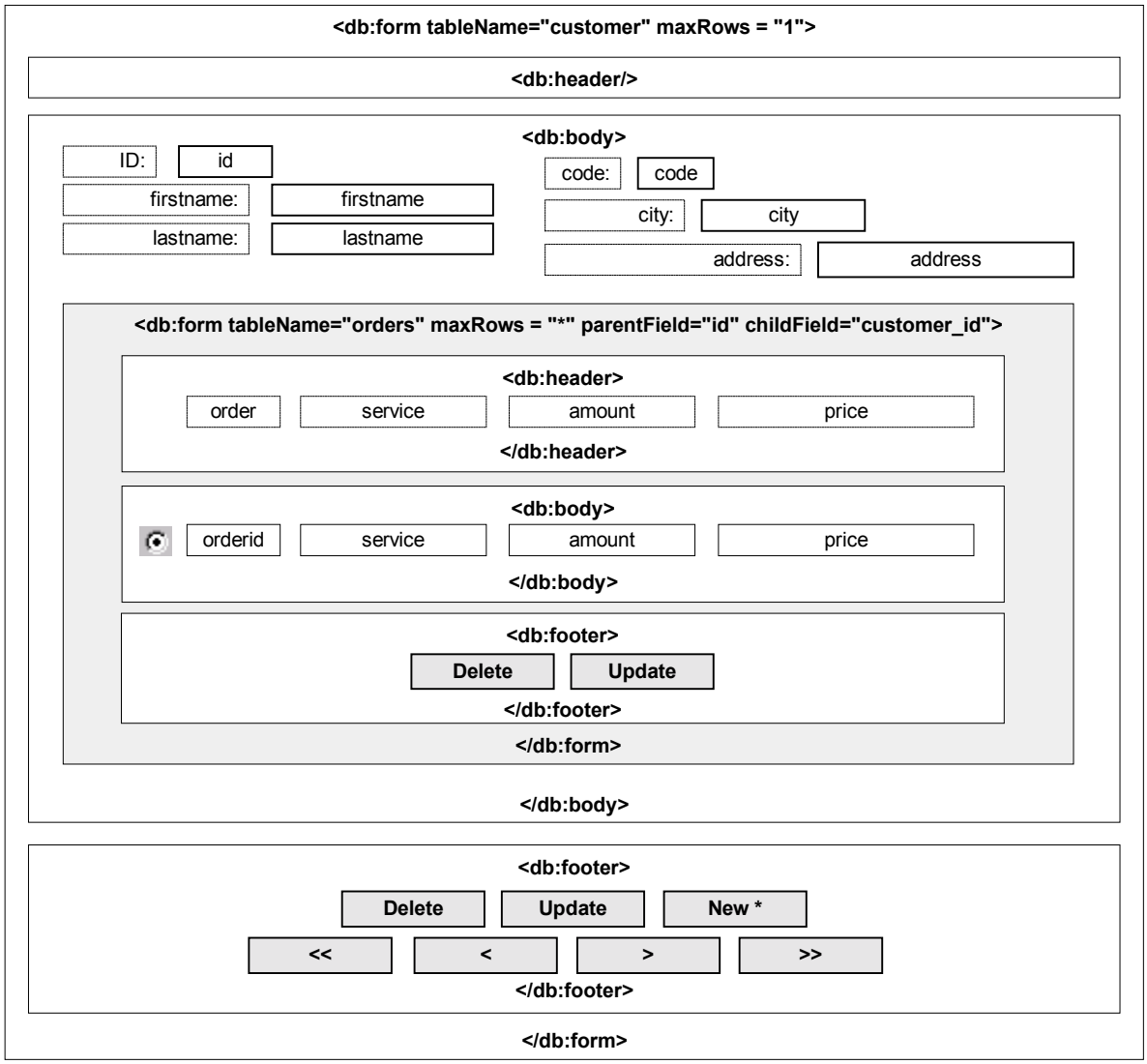

Figure 2 - example of a nested form

The "orders" – form is nested within the "body"-element of the "customer"-form, as shown in Figure 2. The user will see *one* customer per page (because maxRows is set to "1") and *all* the orders (because maxRows = "*") the customer has taken. The user may navigate through the list of customers by clicking the navigation buttons.

3 Ok, let's see some code!

As stated before, the JSP-views of are the only parts of a DbForms application a developer usually needs to put hands on. DbForms provides an extensive custom tag library which makes this an easy task, which can be performed even by non-programmers.

As we will see later, even much of this code can be generated automatically by tools included in DbForms. However, it is useful to understand the basics of DbForms-views, even if much work can be done automatically.

The following two chapters will show a simple and a more advanced example of a DbForms view. Both examples can be thought as being parts of a little CRM-application of a (virtual) agency.

3.1 A simple example

Description:

This JSP-view (service.jsp) enables the user(s) to administer the services the acency provides to its customers. The user (employee of the agency or call center agent) should get a list of all the existing services along with textfields and buttons to update and delete rows. Finally an empty input mask for inserting new services is needed.

service	
PK	**id**
	name description

underlying data

```
<%@ taglib uri="/WEB-INF/taglib.tld" prefix="db" %>

<html>
<head>
  <db:base/>
</head>
<body>
  <db:errors/> <%-- show eventually occured errors --%>

  <db:dbform tableName="service" maxRows="*" followUp="/service.jsp">

   <%-- the header gets rendered one time --%>
   <db:header>
    <db:gotoButton caption="Menu" destination="/menu.jsp" />
    <h1>Services we provide</h1>
      <center><h3>Our existing services</h3></center>
      <table border="5" width="60%" align="CENTER">
        <tr>
          <th>ID</th>
          <th>Name</th>
          <th>Description</th>
          <th>Actions</th>
        </tr>
    </db:header>

    <%-- the body gets rendered for each data row in the query
         it contains textfields and action buttons for manipulating data --%>
    <db:body>
      <tr>
         <td><db:textField fieldName="id" size="5"/></td>
         <td><db:textField fieldName="name" size="20" maxlength="30"/></td>
         <td><db:textField fieldName="description" size="24" maxlength="255"/></td>
         <td>
          <db:updateButton caption="Update"/>
          <db:deleteButton caption="Delete"/>
```

```
        </td>
      </tr>
    </db:body>

   <%-- the footer gets rendered 1 time
        it contains a textfields for entering new datasets --%>
    <db:footer>
      </table>
      <center><h3>Enter new service:</h3></center>

      <table align="center" border="3">
        <tr>
           <td>Id</td>
           <td><db:textField size="5" fieldName="id"/></td>
        </tr>
        <tr>
          <td>Name</td>
          <td><db:textField size="20" maxlength="30" fieldName="name"/></td>
        </tr>
        <tr>
          <td>Description</td>
          <td><db:textArea rows="4" cols="20" wrap="virtual" fieldName="description"/></td>
        <tr>
      </table>

     <br><center><db:insertButton caption="Insert new service!"/></center>
    </db:footer>

  </db:dbform>
</body>
</html>
```

Remarks

- We have set **maxRows** to "*" which has the effect that *all* rows will be shown at once. If there exist hundreds of services we would like to set maxRows to "10", "20" or another "limited" number and we would instanciate navigation – buttons for scrolling between the pages. We will use that pattern later.
- The **<errors/>** tag shows a list of errors, if any occurred (i.e. duplicate key error, etc.)
- The **<db:updateButton>** and **<db:deleteButton>** - tags are placed in the body and therefore rendered for each row.

The result is shown in Figure 3.

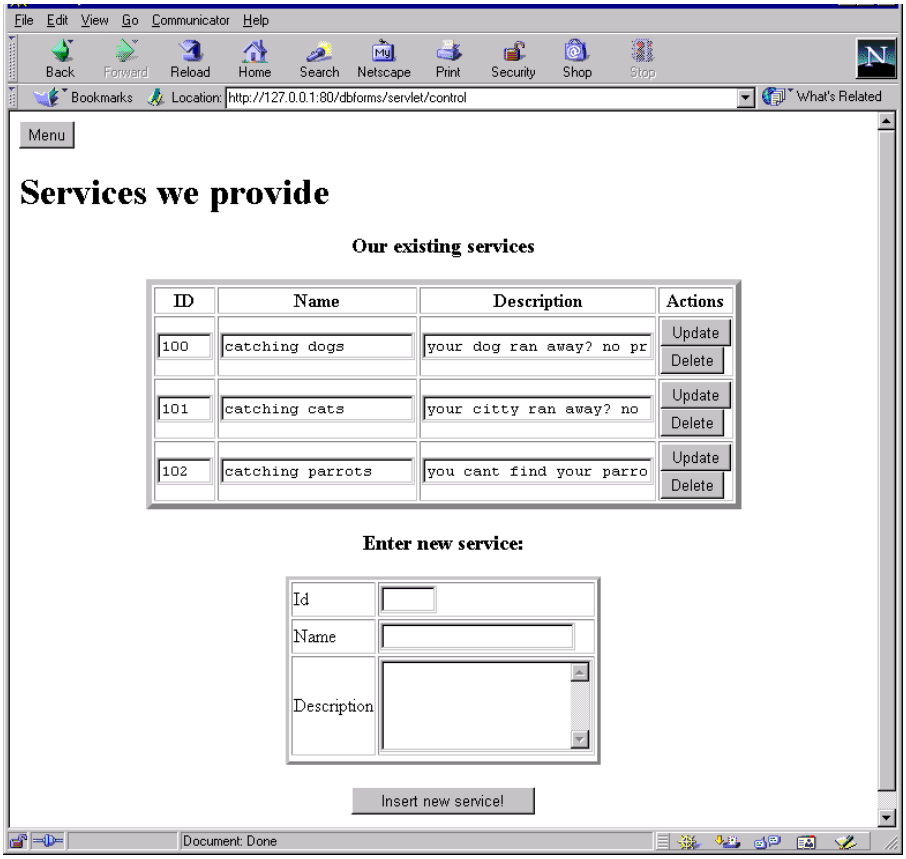

Figure 3 - managing services (service.jsp)

3.2 An example of nested forms

The following page (customer_order.jsp) provides the user the functionality to manage the incoming orders of a customer. The user is able to edit both – orders of a customer – as well as the customer-data itself. Furthermore, the user does not need to struggle with plain service-*IDs*, but he/she able is able to conveniently select the services from a select box *by name*.

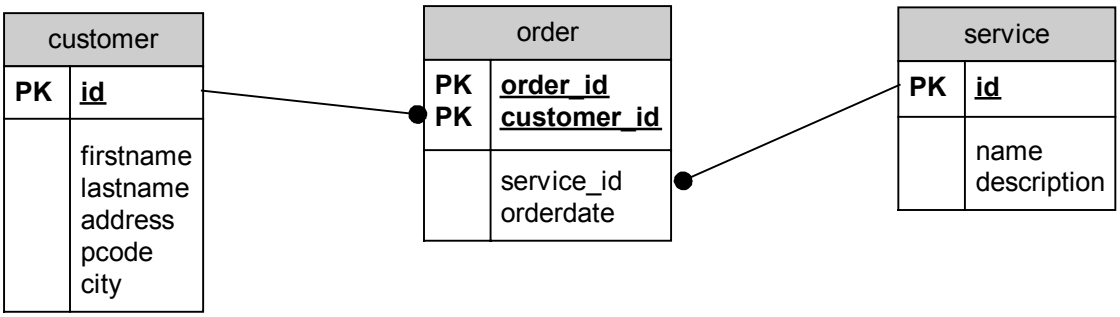

Figure 4 - underlying data for customer_order.jsp

```
<%@ taglib uri="/WEB-INF/dbforms.tld" prefix="db" %>

<html>
<head>
  <db:base/>
</head>
<body>
<db:errors/> <%-- show eventually occured errors --%>

  <%-- the root form --%>
```

```
<db:dbform tableName="customer" maxRows="1" followUp="/customer_orders.jsp"
autoUpdate="false">

 <db:header>
  <db:gotoButton caption="Menu" destination="/menu.jsp" />
  <h1>Customer</h1>
 </db:header>

 <db:body> <%-- the body shows the data of the current customer --%>

  <table align="center">
    <tr>
      <td>Id </td>
      <td><db:textField fieldName="id" size="4"/></td>
    </tr>
    <tr>
      <td>First Name</td>
      <td><db:textField fieldName="firstname" size="18"/></td>
    </tr>
    <tr>
      <td>Last Name</td>
      <td><db:textField fieldName="lastname" size="18"/></td>
    </tr>
    <tr>
      <td>Address:</td>
      <td><db:textField fieldName="address" size="25" /></td>
    </tr>
    <tr>
      <td>Postal code/City</td>
      <td><db:textField fieldName="pcode" size="6"/> -
          <db:textField fieldName="city" size="16"/></td>
     </tr>
  </table>
  <br>

  <%-- table embedding the subform --%>
  <table align="center" border="1">
    <tr>
      <td>
       <center><p><b>Orders</b></p></center>

      <%-- this is the begin of the subform:
           the subform renders all the services the current customer has ordered  --%>
   <db:dbform tableName="orders" maxRows="3" parentField="id" childField="customer_id"
    followUp="/customer_orders.jsp" autoUpdate="false">
      <db:header>
      <%-- code for showing existing orders of services for that customer --%>
      <table>
        <tr>
          <td width="40"></td>
          <td>service</td>
          <td>orderdate</td>
        </tr>
      </db:header>

      <db:body allowNew="false">
        <tr>
          <td width="40"><db:associatedRadio name="radio_order" /></td>
          <td>
            <db:select fieldName="service_id"> <%-- this allows the --%>
              <db:tableData                    <%-- users of the application to --%>
                 name = "our_services"         <%-- select services conveniently --%>
                 foreignTable = "service"      <%-- from a select-box --%>
                 visibleFields = "name"
                 storeField = "id"
              />
            </db:select>
          </td>
          <td><db:dateField fieldName="orderdate" size="14"/></td>
        </tr>
      </db:body>

      <db:footer>

        <tr>
          <td width="40"></td>
          <td><db:updateButton caption="Update Order" associatedRadio="radio_order"/></td>
          <td><db:deleteButton caption="Delete Order" associatedRadio="radio_order"/></td>
```

```
        </tr>
      </table>

      <%-- code for entering new orders of services --%>
      <br><hr>
      <table>
        <tr>
          <td>service</td>
          <td>date</td>
          <td></td>
        </tr>
        <tr>
          <td>
            <db:select fieldName="service_id">
             <db:tableData
              name = "our_services"
              foreignTable = "service"
              visibleFields = "name"
              storeField = "id"
             />
            </db:select>
          </td>
          <td><db:dateField fieldName="orderdate" size="10" /></td>
          <td><db:insertButton caption="insert order" /></td>
        </tr>
      </table>

      <center> <%-- navigating in the subform (in the list of orders of a customer --%>
        <db:navFirstButton caption="&lt;&lt; First" />
        <db:navPrevButton caption="&lt; Previous" />
        <db:navNextButton caption="Next &gt;" />
        <db:navLastButton caption="Last &gt;&gt;" />
      </center>

      </db:footer>
    </db:dbform>
    <%-- subform end --%>

      </td>
    </tr>
    </table>
    <%-- end of table embedding the subform --%>

    <br><center>
      <db:insertButton caption="Store this new Customer!"  />  <%-- action buttons for --%>
      <db:updateButton caption="Update Customer"  />           <%-- editing data of --%>
      <db:deleteButton caption="Delete Customer"  />           <%-- customers --%>
    </center>
  </db:body>

  <db:footer>
     <br><center>
      <db:navFirstButton caption="&lt;&lt; First" />   <%-- navigating in the --%>
      <db:navPrevButton caption="&lt; Previous" />     <%-- list of customers --%>
      <db:navNextButton caption="Next &gt;"  />
      <db:navLastButton caption="Last &gt;&gt;" />
      <db:navNewButton caption”*”/>
     </center>
   </db:footer>
 </db:dbform>
</body>
</html>
```

Remarks

With this page we have demonstrated another couple of major features of DbForms:

- Nested Forms: The structure of this page is similar to the structure shown in Figure 2: a *main*-form has got a *sub*-form inside its body. The sub-form is linked to its parent by the equality of data-fields defined in the child - form's "**`parentField`**" and "**`childField`**" –

attributes [if there are more than one field defining this mapping, a **list** of fields may be provided, with each field separated from the other by the characters like "**,**" or "**;**" or "~"]

- Navigation-Buttons: Because only *one* customer (and *three* orders) is visible at once, the user needs a functionality to navigate between the records. This functionality is provided by **`navFirstButton`**, **`navLastButton`**, **`navPrevButton`**, **`navNextButton`** – elements.

- Insert-Button: the **`navNewButton`** – element navigates the user to an empty form to enter new data.

- Select-Tag: additionally to textField and textArea – tags more complex elements like **`select`**, **`radio`** and **`checkbox`** can be used for data-visualisation and manipulation. I have choosen a **`select`** - tag to select the type of service a customer orders.

- External data fetched by a **`tableData`**-tag: This tag provides external data to **`radio`**, **`checkbox`** or **`select`** - tags. It may be used for cross references to other tables. In our case we initalised the select-box with external data from the table "service".
  Be aware that you have to distinguish between the field(s) to be *shown* to the user and the field to be *stored* in the associated field in the table. In our case we *shown* the field "service.name" and *stored* the value "service.id" in the associated field "orders.service_id"! The name "**`our_services`**" was defined to enable internal caching of the data, which increases performance.

- Using `associatedRadio`-elements to mark rows of data for certain actions (in our case we mark "orders" for actions "update" and "delete") saves a lot of space and makes the interface clearer. If we had to draw a button for all possible actions, the page would not look very friendly.

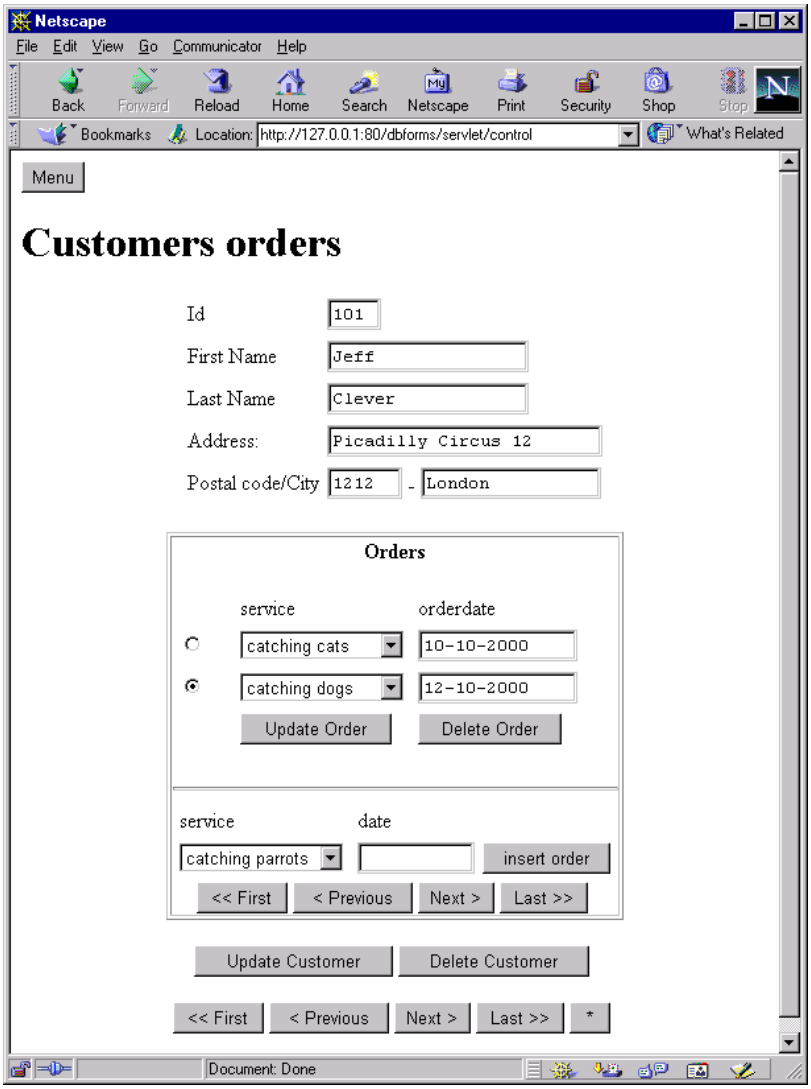

Figure 5 - managing orders and customers with one single page (customer_order.jsp)

4 Advanced Features

4.1 Fine grained security

DbForms' security model builds on top of the Java Servlet security model, with its concept of users (principals) and roles.

DbForms provides fine-grained declarative definition of rights for data-access and –manipulation. DbForms allows to attach security constraints to each table defined in the xml-configuration telling DbForms which kind of database operations may be executed by which user groups.

```
<dbforms-config>

  <table name="customer" >
    <field name="id" fieldType="int" isKey="true" />
    <field name="firstname" fieldType="char" />
    <field name="lastname" fieldType="char" />
    <field name="address" fieldType="char" />
```

```
    <granted-privileges
      select = "A,B"
      insert = "A"
      update = "A,B"
      delete = "A" />
  </table>

</dbforms-config>
```

Listing 2 – defining privileges

The attributes of the <granted-privileges> element tell DbForms: "Members of group *A* may select, insert, update and delete customers, members of *B* may read and update customers". All other groups (for example a group *C*) may not access this table at all.

4.2 File Uploads

Managing BLOB Fields is a very easy task when using DbForms: First you have to tell DbForms about BLOB-Fields in the xml-configuration file:

```
<dbforms-config>

  <table name="pets">
    <field name="pet_id" fieldType="int" isKey="true" autoInc="true" />
    <field name="name" fieldType ="char" />
    <field name="portrait_pic" fieldType ="blob" />
    <field name="story" fieldType ="blob" />
 </table>

</dbforms-config>
```

Listing 3 – defining fields of type "BLOB"

The configuration-code-snippet shown above in Listing 3 tells DbForms that the fields "portrait_pic" and "story" are BLOBs. As you can see, DbForms allows more than one field in a row to be a BLOB.

After defining our BLOB-powered table, we would want to build a JSP for managing the BLOB fields. For this purpose a new custom tag is introduced:

```
<db:file fieldName="portrait_pic">
```

Listing 4 – Implementing a file tag

The attribute `fieldName` refers to the name of the field the file needs to be uploaded to. (There exist additional attributes available for this element not shown here.)

This custom tag gets rendered as HTML <input type=”file”> - tag, as shown in Figure 6

Figure 6 – the result visible to the user

This HTML-element enables multipart-enabled browsers to submit files to the server.

If where using BLOBs for storing images in a database, could write the following JSP-code to *retrieve and render* such a field:

```
<img src=”<db:blobURL fieldName="portrait_pic"/>”
       width=”100” height=”80” border=”0”>
```

Listing 5 – rendering images using a blobURL

Special DbForms-feature: DISKBBLOBs

There are situations where BLOBs are not an option (if the application uses a RDBMS or JDBC-driver without BLOB-support, or if BLOB-support is too slow or even buggy, or if the files should be accessible by other applications without using a database layer)

DbForms also manages uploads to a file system instead of a database. This is completely **transparent** to the JSP (View) – developer! For uploading and retrieving file-system-stored objects **the same tags and attributes** are used as for uploading and retrieving regular BLOBs.

The only difference is in the definition of the *Model*, where a server-directory for storing the files must be specified additionally.

```
<field name="story" fieldType ="diskblob" directory=”x:\stories” />
```

Listing 6 – defining fields of type “DISKBLOB”

4.3 Sorting, searching and filtering

4.3.1 Sorting

Sorting is an important, even essential, feature for many database applications. For example, if we would like to give the user the ability to sort the virtual agency’s services by id, then we would use the following DbForms “sort” – tag:

```
<db:sort fieldName="id" />
```

Listing 7 – instantiating a sort - tag

4.3.2 Filtering

Just like sorting, filtering of data rows is an important, even essential, functional requirement for many database applications.

In its simplest case, the "Filter" is nothing more than a part of the WHERE clause of the SQL SELECT statement used to retrieve the rows.

The filter criteria may be passed to the "filter" attribute of the dbform tag:

```
<db:dbform tableName="employee" maxRows="*" followUp="/employees.jsp"
autoUpdate="false" filter="deptno=101 AND salery>3000"
...
</db: dbform>
```

Listing 8 – A example for static filtering

4.3.3 Searching

Searching is another "must-have" functionality for a database application framework like DbForms.

DbForms allows you to create search-forms very quickly. The number of fields to be searched, the kind of input widget (textfield, textArea, select box, etc.), the kind of search algorithm to use and the Boolean combination of matches of search criteria are completely flexible.

All you have to do is

- to decide which fields you want to make searchable
- to decide if you want the user to choose the criteria-combination-mode himself/herself or if you want to use a (hidden) default
- to decide if you want the user to choose the kind of search algorithm to be used himself/herself or if you want to use a (hidden) default

Figure 7 shows a fully functional search-panel. It is included in the example coming along with the DbForms distribution. You may also test it on the running live samples at the DbForms Website.

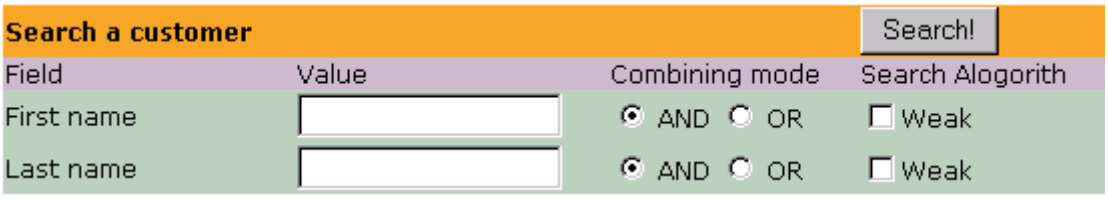

Figure 7 –Example for dynamic filtering

4.4 Implicit Scriping Variables

DbForms tags like <db:dbform> and <db:body> make some information available to the embedded JSP code where the values are accessible as simple **scripting variables**

The mechansim used for passing these values over is called "**Tag Extra Info**" and is part of the Java Server Pages specification (compare specification version 1.1, chapter 5.5)

Two important scripting variables are listed in Table 1:

`currentRow`	`java.util.Hashtable`	Contains the field-values of the current row. This construct is similar to "associated Arrays" used in many Perl/PHP modules `Example:` **`String email = (String) currentRow.get("email");`** Scope: inside the respective <body>-element
`position`	`java.lang.String`	Contains the encoded key-fieldvalues of the current row. Scope: inside the respective <body>-element

Table 1 – two useful DbForms scripting variables

These scripting variables can be used by a JSP developer to add advanced functionality to her/his DbForms-driven JSP page.

4.5 Application Hookups

4.5.1 Introduction

DbForms offers much built-in functionality as well as many optional features, which may be configured and utilised using the XML configuration file, attributes of custom tags, etc.

But it would be neither possible nor useful to create a system which has got a built-in hardcoded answer for every problem or requirement which could appear during the development-process (and life-cycle) of a database application. Because it is impossible to foresee all eventual use cases and user's needs, such a monolithic approach would certainly restrict the application developer sooner or later.

4.5.2 Interface DbEventInterceptor

Every system which wants to get around this problem has to offer a kind of "programming facility" or "Application Programming Interface". So does DbForms:

DbForms provides an Interface "`DbEventInterceptor`" which intercepts database operations DbForms is about to perform or has finished. This interface provides the following methods:

```
public int preInsert(HttpServletRequest request, Hashtable fieldValues, DbFormsConfig
  config, Connection con)
throws ValidationException;

public void postInsert(HttpServletRequest request, DbFormsConfig config, Connection con);

public int preUpdate(HttpServletRequest request, Hashtable fieldValues, DbFormsConfig
  config, Connection con)
throws ValidationException;;

public void postUpdate(HttpServletRequest request, DbFormsConfig config, Connection con);
```

```
public int preDelete(HttpServletRequest request, Hashtable fieldValues,DbFormsConfig
  config, Connection con)
throws ValidationException;;

public void postDelete(HttpServletRequest request, DbFormsConfig config, Connection con);

public int preSelect(HttpServletRequest request, DbFormsConfig config, Connection con)
throws ValidationException;;

public void postSelect(HttpServletRequest request, DbFormsConfig config, Connection con);
```

Listing 9 – methods definied in interface DbEventInterceptor

As the names indicate, the preXxx() methods get called **before** the respective database operation is performed, the postXxx() methods get called **after** the operation was finished.

PreXxx() methods return a value indicating if the operation should be performed or not

- `DbEventInterceptor.GRANT_OPERATION`
- `DbEventInterceptor.DENY_OPERATION`

PostXxx() methods do not return a value, as the operation is already done.

4.5.3 Installing the Hookups

How do we tell DbForms *when* to invoke *which* Interface implementation?

This information must be provided in the DbForms XML configuration file. Similar to the "granted-privileges" security-constraint (described above) the XML tag defining an Interceptor has to be placed inside a <table> element.

```
<table name="customer">
  <field name="id" fieldType="int" isKey="true"/>
  <field name="firstname" fieldType="char" />
  <field name="lastname" fieldType="char" />
  <field name="address" fieldType="char" />
  <field name="pcode" fieldType="char" />
  <field name="city" fieldType="char" />

  <interceptor
    className = "com.foo.bar.CustomerValidatonChecker"
  />

  <interceptor
    className = "com.foo.bar.TransactionLogger"
  />
</table>
```

The semantics of these declarations could be described as *"Invoke `com.foo.bar.CustomerValidatonChecker` and `com.foo.bar.TransactionLogger`, if the user is about to read, insert, update or delete data from table `customer` and call the appropriate methods of those objects"*

5 Rapidly generating standard-views using XSL

As demonstrated in the last chapters, DbForms can help speeding up development of web based database applications. But there are still lots of monotonous tasks to do - you **still** have to write the **JSP code for each view**. The more views you have to write, the more monotonous work you have to do.

Because the JSPs are very similar to each other in many cases, you'll probably do much of the work using copy-and-paste and then apply the changes necessary by hand. This works well, but do we really want this kind of development? I don't think so!

For instance, if you have got a database containing 40 tables and you want to write a web based data maintenance tool against it. Imagine you want for each table a "list-view" (allowing the users to view all elements of the table at one time) and a "single-view" (viewing only 1 row at a time) so that the user may view and edit the data in 2 different ways. Well, having 40 tables and 2 views for each, implicates that you will have to create 40x2 = 80 JSP files, and an application-menu as well. This means that at least 81 JSP files need to be written!

Thanks to XML and XSL DbForms provides a pretty simple and straightforward solution:

- For every "standard-layout" (in our application we have got "*listview*-layout", "*singleview*-layout" and "*menu*-layout") we create **one XSL file**. Each XSL file gets applied to your application's XML-configuration file (this file contains detailed information about all tables and fields of the database)

- The result of the XSL transformations are JSP files, created according to the style rules defined in the XSL file.

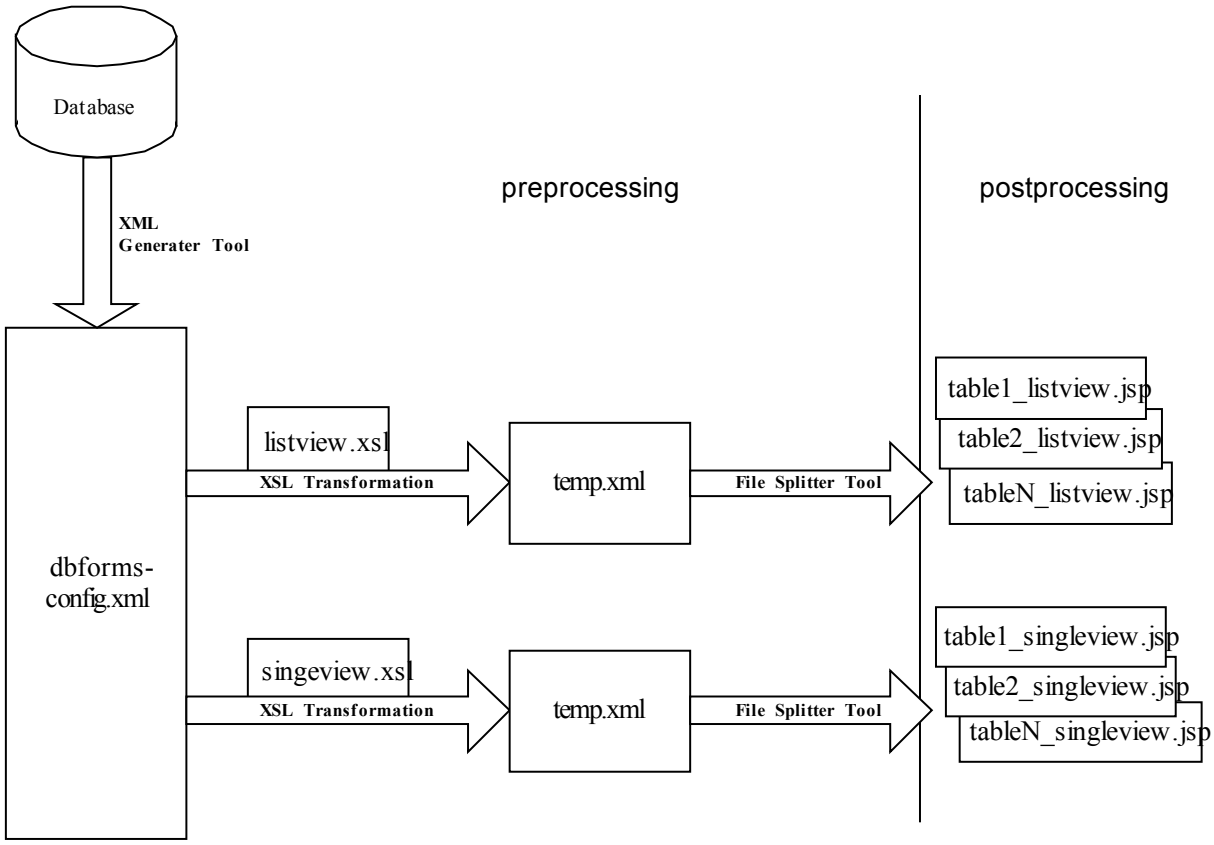

Figure 8 – chain of transformations that builds a DbForms-application – in the example shown two stylesheets are used. (The number of stylesheets is unlimited of course)

The very good news is that the whole process shown in Figure 8 is encapsulated by a SWING-based Tool called "**DevGui**", shown in Figure 9. You can use this application to generate the XML formatted - database metadata, to apply XSL-stylesheets to it, and to deploy and test the resulting JSP files.

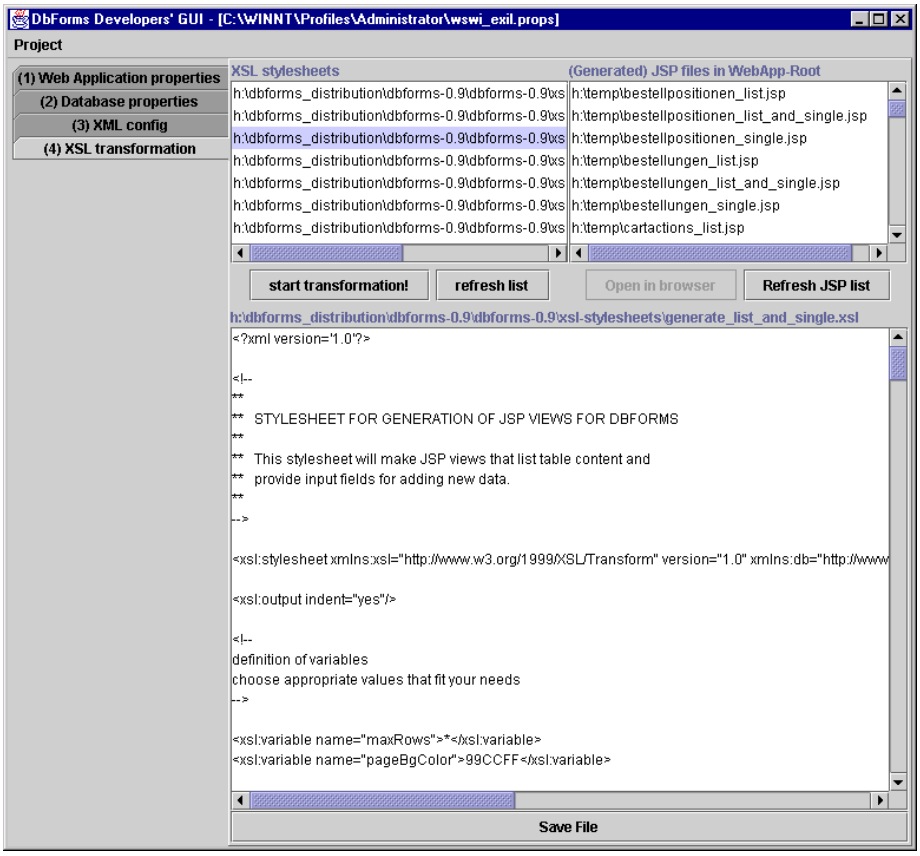

Figure 9 – DevGui: a tool for automatically generating JSP views

Appendix - DbForms Custom Tag Library

Tag Name	Description
dbform	a database application form (root element of a JSP view)
header	this tag renders a Header-tag. It should be nested within a dbform-tag
body	this tag renders a Body-tag. It should to be nested within a dbform-tag.
footer	the footer grouping tag. it is supposed to be nested within a dbform-tag
label	this tag renders a database-data-driven label, which is a passive element (it can't be changed by the user) - it is predestined for use with read-only data (i.e. primary keys you don't want the user to change, etc)
dateLabel	Similar to “label” but with special attributes for formatting date values
dataLabel	label - a tag for data presentation, not for editing data (=> can never be used as input field)

textField	this tag renders a dabase-datadriven textfied, which is an active element - the user can change data
dateField	this tag renders a dabase-datadriven date-field, which is an active element - the user can change data. it is very similar to textField but it has go special attributes for formatting date values
textArea	renders a HTML textarea element
tableData	external data to be nested into radio, checkbox or select - tag! (useful only in conjunction with radio, checkbox or select - tag)
queryData	external data to be nested into radio, checkbox or select - tag! (useful only in conjunction with radio, checkbox or select - tag)
staticData	external data to be nested into radio, checkbox or select - tag! (useful only in conjunction with radio, checkbox or select - tag)
staticDataItem	data to be nested into staticData element
file	this tag renders an upload-button for uploading files into BLOBs or DISKBLOBs
radio	renders a HTML radio-element or a whole group of them
checkbox	renders a HTML checkbox-element or a whole group of them
select	This tag renders a html select element including embedded option-elements.
linkURL	generates a link to a DbForms View
position	element to be embedded inside a linkURL-element
insertButton	this tag renders an insert-button
deleteButton	this tag renders a delete-button
updateButton	this tag renders an update-button
navPrevButton	this tag renders a navigation button for scrolling to previous row(s) of the current table
navNextButton	this tag renders a navigation button for scrolling to next row(s) of the current table
navFirstButton	this tag renders a navigation button for scrolling to the first row(s) of the current table
navLastButton	this tag renders a navigation button for scrolling to the last row(s) of the current table
navNewButton	this tag renders a navigation button for creating a new row of data
gotoButton	Renders a button for jumping to a JSP-view
base	renders a HTML-base tag
errors	Custom tag that renders error messages
associatedRadio	This tag enables the end-user to define a row by selecting the radio-button rendered by this tag
blobURL	This tag generates an URL pointing to a servlet downloading a

	binary object
sort	this tag renders a select box for switching the order-state of a field (ascending, descending, none)
style	generic style tag

References

[Serlvet] http://java.sun.com/products/servlet/
[JSP] http://java.sun.com/products/jsp/
[Taglib] http://java.sun.com/products/jsp/taglibraries.html
[Gamma] Erich Gamma, Richard Helm,Ralph Johnson, and John Vlissides “Design Patterns: Elements of Reusable Object-Oriented Software” Addison Wesley. October 1994.

Author

Joachim Peer is an independent J2EE developer. He holds a masters degree in Management Information Systems of Johannes Kepler University, Linz, Austria, with a focus in the fields of software engineering, multimedia application development and knowledge and process management. He is a Sun certified programmer for the Java 2 platform.