MDV technical

Table of Contents

Intro

This document describes the implementation of MDV. It is intented for programmers and integrators who would like to have a technical introduction to the system.

This document is included in OpenMDV and written in DocBook. The source can be found in MDV’s directory src/resources/mdv/xdocs/DOC-techguide.xml

Prerequisites

Java SDK. Get it from java.sun.com. Standard versions 1.3 and 1.4 are ok.

Get the openmdvp (mind you, not openmdv) source code from cvs (cvs co openmdvp). Please check mdv.sourceforge.net for details.

Building the installation from sources: in your openmdvp directory, issue: sh build.sh createinstallationjar (in unix systems) or build createinstallationjar (in MS Windows systems). After a while, your installation package openmdvp.jar is ready in your openmdvp directory. Now you can write changes to the source code and check out how they change the behaviour of the system, as follows:

- hack some code,
- build createinstallationjar
- If you have openmdvp running (from previous iteration), stop tomcat (jakarta-tomcat-4.0.1/bin/catalina.sh stop) and remove at least jakarta-tomcat-4.0.1/webapps.
- (Re-)install openmvdp.jar: java -jar openmdvp.jar

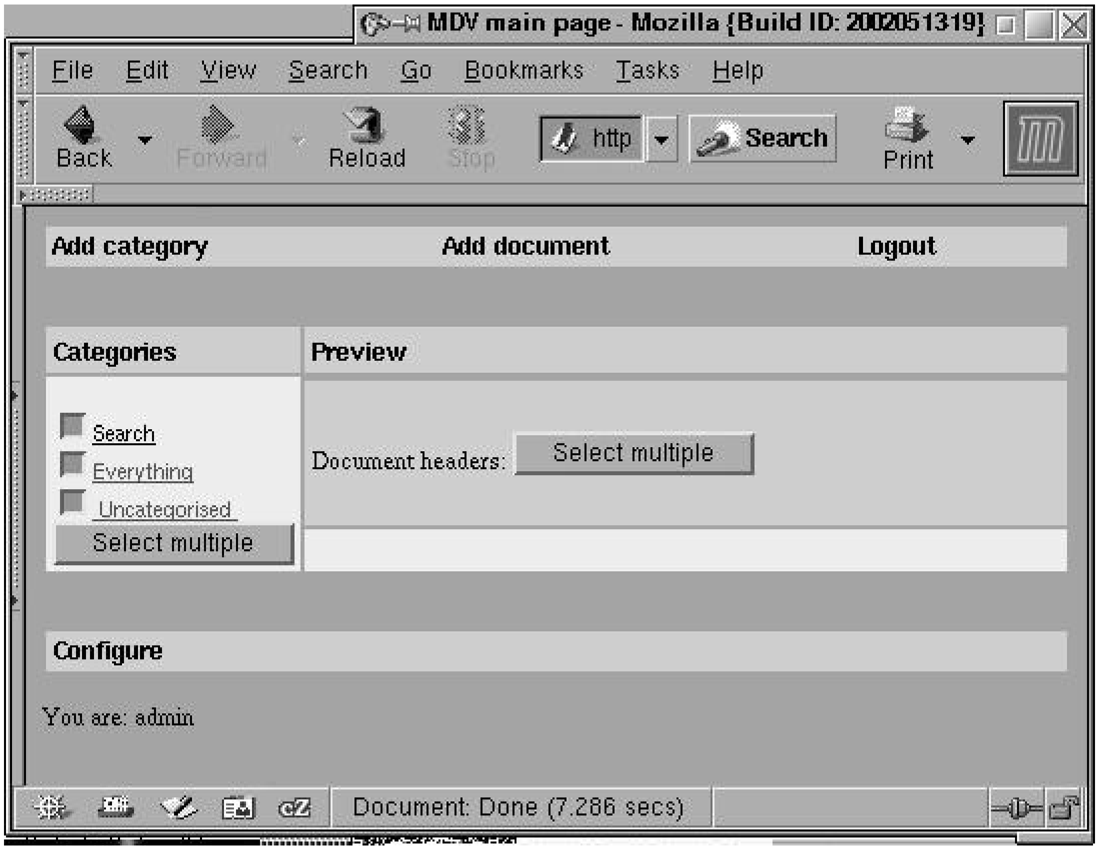
How does it work?

To put it briefly, MDV is a collection of java classes and template files that are run under control of COCOON servlet inside of Tomcat servlet container.

This enables us to make a distinction between

- storages that contain data,
- domain objects, i.e. objects of java classes that represent classes of objects in the real world;
- the representation of domain objects in XML;
- the user interface, i.e. HTML or some other "rendering" of XML representation of (some of the) domain objects and actions available to the user;

This may sound complicated, but the following example will illustrate the functionality. The basic domain object classes of "out-of-box" MDV are Document, Category and Principal (User or Group). For each one of them, XMLConverter can produce an XML representation. Suppose a user called Admin logs in. This is what he sees:


MDV main page - Mozilla {Build ID: 2002051319}
File Edit View Search Go Bookmarks Tasks Help
Back Forward Reload Stop http Search Print
Add category
Add document
Logout
Categories
Search
Everything
Uncategorised
Select multiple
Preview
Document headers:
Select multiple
Configure
You are: admin
Document: Done (7.286 secs)

This is a view generated by XSL templates out of XML source that looks like this:

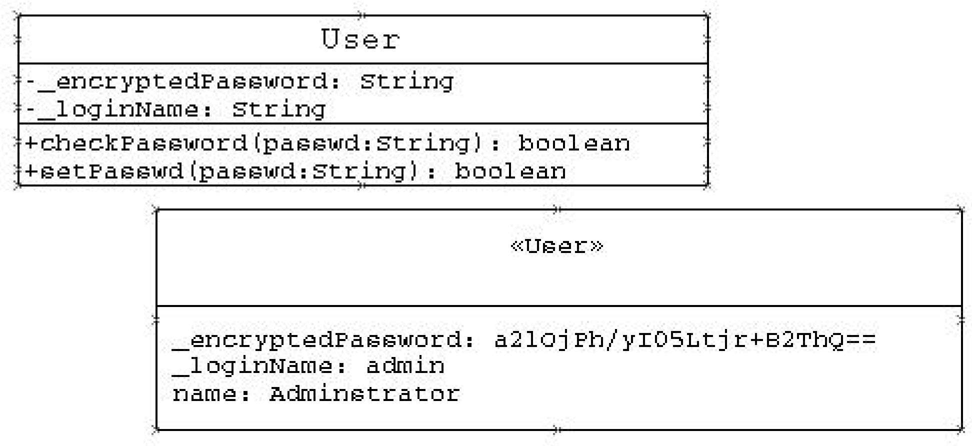
```
<result>
 <tree>
  <treehead>
   <treerow>
    <treecell label="category.name.root" id="1"/>
   </treerow>
  </treehead>
  <treechildren>
   <treeitem container="false" open="false" selected="false">
    <treerow>
     <treecell label="category.name.search" id="1.1" />
    </treerow>
   </treeitem>
   <treechildren>
    <treeitem container="true" open="false" selected="false">
     <treerow>
      <treecell label="category.name.builtin" id="1.2" />
     </treerow>
    </treeitem>
    <treeitem container="false" open="false" selected="false">
     <treerow>
      <treecell label="category.name.all" id="1.2.1" />
     </treerow>
    </treeitem>
    <treeitem container="false" open="false" selected="false">
     <treerow>
      <treecell label="category.name.uncategorized" id="1.2.2" />
     </treerow>
    </treeitem>
   </treechildren>
   <treeitem container="false" open="false" selected="false">
    <treerow>
     <treecell label="category.name.user" id="1.3" />
    </treerow>
   </treeitem>
```

```
    </treechildren>
   </tree>
  <user ID="1" owner-iD="1">
   <name>Administrator</name>
   <password>a2lOjPh/yI05Ltjr+B2ThQ==</password>
   <login-name>admin</login-name>
  </user>
  <documentlist></documentlist>
  <selected-docs><documentlist></documentlist></selected-docs>
 </result>
```

The XSL templates parse this representation in order to create a "pretty" HTML rendering of it for the user. We see that most of XML representation is dedicated to the tree of categories, shown on the left. On the bottom, you see the user id, and in the XML listing, of course, the corresponding XML entry. There are no documents in any of the selected categories (actually, none is selected) so, in the XML representation both documentlist and selected-docs are empty.

What about actions? Based on the objects that are selected (categories, documents), the menu bar on top shows the actions that are available. Here, only "Add category", "Add document" and "Logout" are available.

The information that is represented using XML has to come from somewhere. As you guessed, it comes from objects. As an example, here’s a java class (in UML) User and the corresponding entry for user Admin (attribute name has been inherited from the parent class).


User
-_encryptedPassword: String
-_loginName: String
+checkPassword(passwd:String): boolean
+setPasswd(passwd:String): boolean
«User»
_encryptedPassword: a2lOjPh/yI05Ltjr+B2ThQ==
_loginName: admin
name: Adminstrator

In the "out-of-the-box" installation, the data is eventually stored in a relational database, and retrieved from there. However, the storage can be anything that can be interfaced by java "dataio" classes.

The WAR file -- openmdvp web application

In this section, we’ll take a closer look of what files there are in openmdvp package (mdv.war). Mdv.war is a standard war file that contains a web application. It is deployed by Tomcat server during the startup of the server.

WEB-INF/lib -- here are the jar files needed to run the system, many of them are related to database access and XML parsing. Mdv-web.jar and mdv-dataio.jar are, however, the core of mdv system. For details, see next section.

WEB-INF/db -- the database is here. It is accessed through port 9001 by default.

xstyles -- these are docbook stylesheets, needed to display docbook documentation.

xdocs -- documentation in docbook format.

sitemap.xmap -- the core of cocoon’s action processing.

protected/xstyles, protected/xdocs, protected/sitemap.xmap: these are the XML machinery of the MDV application. The xsp files in protected/xdocs contain short java extracts in order to output "just the right amount" of XML to the corresponding XSL stylesheets (in protected/xstyles). protected/sitemap.xmap controls this by coupling together X.xsp with X.xsl.

Openmdv classes in mdv-dataio.jar and mdv-web.jar

In http://mdv.sourceforge.net you'll find a complete annotated UML class diagram of the dataio classes, so only the main ones are explained here. Roughly, mdv-dataio is the structure of the application, complete with possibilities to generate XML to the "outside world" of the user agents and the "inside world" of storage systems. However, it contains nothing that takes care of the flow of information to the user or reacts when the user clicks some link or submit button. These are handled by mdv-web ("web tier").

mdv-dataio, the structure

fi/hip/mdv/dataio contains abstract dataio classes. The implementing dataio classes are e.g. in fi/hip/mdv/dataio/rdb (relational db access). DataAccess is a "portal" class that provides the methods to use most other classes, i.e. it contains methods for login, reading principals, documents, categories, saving and removing them etc. Please see the javadoc documentation of this class for details.

CompoundDataAccess implements DataAccess. Using CompoundDA, it is possible to access several storages simultaneously (or actually sequentially). In the out-of-the-box installation CompoundDA is the class that the web tier talks to. There, it has only one member of the compound, and that's the rdb data access that communicates with the local database.

DocumentType is an interface that models Document's type information. DocumentType has a name and Fields that the documents of this type contains.

Document and Field are quite easy to understand on the basis of DocumentType. FieldIO is a way be which different fields store their data. Schema is an auxiary class for fast access of Document and Field data.

Principal, User and Group are classes to handle user and group data.

mdv-web, the action driver

The session/MDV*Action classes are an interface to HTML forms with an action parameter e.g.

```
<input type="hidden" name="action" value="categorize"/>
```

The basic function of the MDV*Action classes is to select the correct translation handler based on the action and then call the handler.

A handler provides two main methods, syncGuiToModel and syncModelToGui, and a web interface. There are handlers for many, if not all, classes that can have some interactions with the user (Category, Document, FieldValue, etc).

syncGuiToModel gets the stored information of the object from the storage i.e. storage to object. syncModelToGui stores the information contained in the object into the storage i.e. object to storage.

WebInterface provides methods for setting the fields of the object that is being processed. Here, the strings of the web forms are "translated" into other scalar types, for instance longs. For instance, a document type has fields that have numeric id's. The web interface can only provide strings, so they are converted in setFieldIds method.

The following section gives a concrete example of the flow of information when using dataio and web tier.

The flow of control in practice

From dataio to user's browser:

The user authenticates himself as admin and arrives on the main screen. The URL is protected/protected.

As shown in protected/sitemap.xml, this URL launches the content in protected.xsp and this content is filtered through page-html.xsl

```
<map:match pattern="protected*">
<!-- first validate whether user has logged in -->
<map:act type="mdv-session-validator">
<!-- generate protected content -->
<map:generate type="serverpages" src="xdocs/protected.xsp"/>
<map:transform src="xstyles/page-html.xsl"/>
<map:serialize/>
</map:act>
<!-- something was wrong, redirect to login page -->
<map:redirect-to uri="login"/>
```

```
          </map:match>
```

protected.xsp contains a xsp:logic section that retrieves all the categories accessible to the user, as well as user information and selected documents. Here’s an extract of calling dataio’s user information function:

```
          u = mdvDA.getCurrentUser();
```

All this is collected (in protected.xsp) in a string, that contains XML. Here’s how the user part is included in the string:

```
     result = result + XMLConverter.Convert(u);
```

This string is parsed in the last part of protected.xsp:

```
        Parser newParser = null;
        try {
                newParser = (Parser) this.manager.lookup(Parser.ROLE);
                InputSource is = new InputSource(new StringReader(result));
                XSPUtil.include(is, this.contentHandler, newParser);
        } catch (Exception e){
              System.out.println("Exception in xsp"+e);
        }
        finally {
              this.manager.release((Component) newParser);
        }
```

The parsed output is sent to page-html.xsl, as defined in sitemap.xsp. The following extract of page-html.xsl generates html out of user information by calling another template:

```
        <xsl:call-template name="user"/>
```

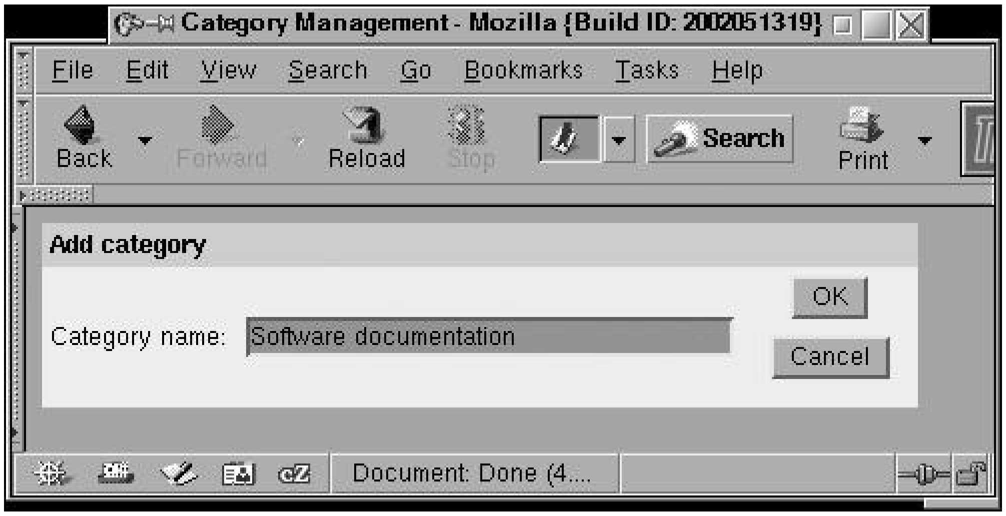
Template "user" is in page-styles.xsl and looks like this:

```
          <xsl:template name="user">
           You are: <xsl:apply-templates select="result/user/login-name"/>
          </xsl:template>
```

Naturally, once html has been generated, tomcat serves it to the browser.

From user’s browser to dataio:

Let’s assume the user wants to create a new category, as in the following figure:


Category Management - Mozilla {Build ID: 2002051319}
File Edit View Search Go Bookmarks Tasks Help
Back Forward Reload Stop Search Print
OK
Cancel
Document: Done (4....

Add category

Category name: Software documentation

The corresponding form (some formatting omitted) is:

```
<form action="mdv-add-category" method="post">
<input size="40" name="setName" type="text">
<input value="add" name="action" type="hidden">
<input value=" OK " type="submit">
</form>
```

In sitemap.xmap we find the following mapping that binds the URL mdv-add-category with destinations mdv-act-category (if everything is ok), login (if the user is not logged in) or CATEGORYNEW (if the user gave an empty name).

```
<map:match pattern="mdv-add-category">
 <map:act type="form-validator">
  <map:parameter name="descriptor"
   value="context://protected/descriptors/params.xml"/>
        <map:parameter name="validate" value="setName"/>
        <!-- then validate whether user has logged in -->
        <map:act type="mdv-session-validator">
                <map:act type="mdv-act-category">
                <map:redirect-to uri="protected"/>
        </map:act>
                <map:redirect-to uri="login"/>
        </map:act>
                        </map:act>
                        <!-- value for the category name failed,
                        let’s ask it again.
                        -->
                        <map:redirect-to uri="CATEGORYNEW"/>
  </map:match>
```

Since the name is ok and the user is logged in, we end up in action mdv-act-category. This is mapped with class MDVCategoryAction, as shown in sitemap.xmap.

```
<map:actions>
..
<map:action name="mdv-act-category"
src="fi.hip.mdv.web.acting.MDVCategoryAction"/>
..
</map:actions>
```

MDVCategoryAction is usually only interested in the action parameter. Since it is add, MDVCategoryAction gets a handler for the action, as follows:

```
Translator trans = mdvSession.getTranslator();
TranslationHandler handler =
            trans.getHandlerFor(CategoryTranslationHandler.class, action);
```

After that, MDVCategoryAction iterates all its parameters using the handler:

```
            Enumeration enum = req.getParameterNames();
            while (enum.hasMoreElements()) {
                 String param = (String) enum.nextElement();
                 String[] paramValues = req.getParameterValues(param);
                 Object result = handler.translate(param, paramValues);
         }
```

What happens with the "translate" call is taken care by the translator, CategoryTranslationHandler. Let’s recall that the parameter was setName. It is not a coincidence that CategoryTranslationHandler’s web interface has a method by the same name -- it’s by design. After the handler.translate -call CategoryTranslationHandler "knows" the name of the new category, and because of the action parameter (add), it knows that a new category with that name will be created.

The creation of the new category is ordered by MDVCategoryAction:

```
             handler.syncModelToGui(handler.getRealSubject(),mdvSession);
```

This call in the handler means:

```
         if (o instanceof UserCategory)  uc = (UserCategory) o;
         ...
         int savedID = da.saveCategory(uc);
```

Thus, the new category is saved and the user redirected to the main page (protected.xsp).

Summary and advice for developers

If you want to change how things look, modify XSL files in src/resources/protected/mdv/xstyles.

If you want to change how pages/states change (i.e. how to move from one page to another via some user action), change src/resources/protected/mdv/sitemap.xmap

If you want MDV to handle completely different things (calendar data, e-mails, etc), derive new document/field/category classes and possibly a DataAccess to access different kinds of storages (ICAP, IMAP, etc).

Acknowledgements

This guide was inspired by Maiju Virtanen’s guide to the previous version of MDV.