# ComposerMVC User Guide

**By Paul Done**

# 1. Introduction

ComposerMVC is Model-View-Controller (MVC) framework for use in exteNd Composer projects.

Typically Novell's exteNd Composer product is used by developers to create web services which represent an organisation's processes. These services usually encapsulate data and logic which already exist in one or more back-end systems.

The user interface for Composer developed web services are normally constructed using other products and programming languages. These products are used to consume the Composer web services. For example: an exteNd Director portal, a C# .NET GUI application, or a J2EE web application.

Since version 5.0, exteNd Composer has provided native capabilities for creating simple HTML interfaces using the Java Server Pages (JSP) technology. However, on its own this capability is fairly limited and, if used, forces the developer to embed large sections of Java code (scriptlets) within the Composer JSPs. Validation, inter-page navigation and linking is also very difficult to achieve using Composer JSPs.

The ComposerMVC framework introduces the MVC design pattern (http://java.sun.com/blueprints/patterns/MVC.html) to a Composer project to try to eliminate some of these shortcomings. A ComposerMVC project can be divided into the three logical sections:

- ***Model*** : the normal Composer Web Services that integrate with back-end systems;
- ***View***: a set of standard Composer JSPs displaying a mix of static HTML and dynamically generated HTML (via the standard Composer Tag Library);
- ***Controller***: a special 'container' JSP, a special 'controller' web service and a set of 'action' XML Map components.

The benefits of the ComposerMVC framework are:

- developers build web services in the normal way, enabling these web services to subsequently be used remotely and unchanged if a future decision is made to move to a full user interface toolkit such as exteNd Director later on in a project's lifetime;
- normal JSPs contain no scriptlet code and only use HTML, standard JSP syntax and Composer's standard tag library to dynamically generate pages, thus reducing the overhead of page creation and maintenance;
- all validation logic for HTML Form submission, calls to update or query the model and navigation logic to determine what view to display, is implemented in individual 'action' components, where programmatic logic is most easily accomplished in Composer.

The ComposerMVC framework is NOT a replacement for a fully fledged user interface toolkit such as exteNd Director. Ideally, such toolkits should be used given sufficient skills, resources and licenses. However this is not an ideal world, hence this framework can provide a service to a development team which falls into one of the following three categories:

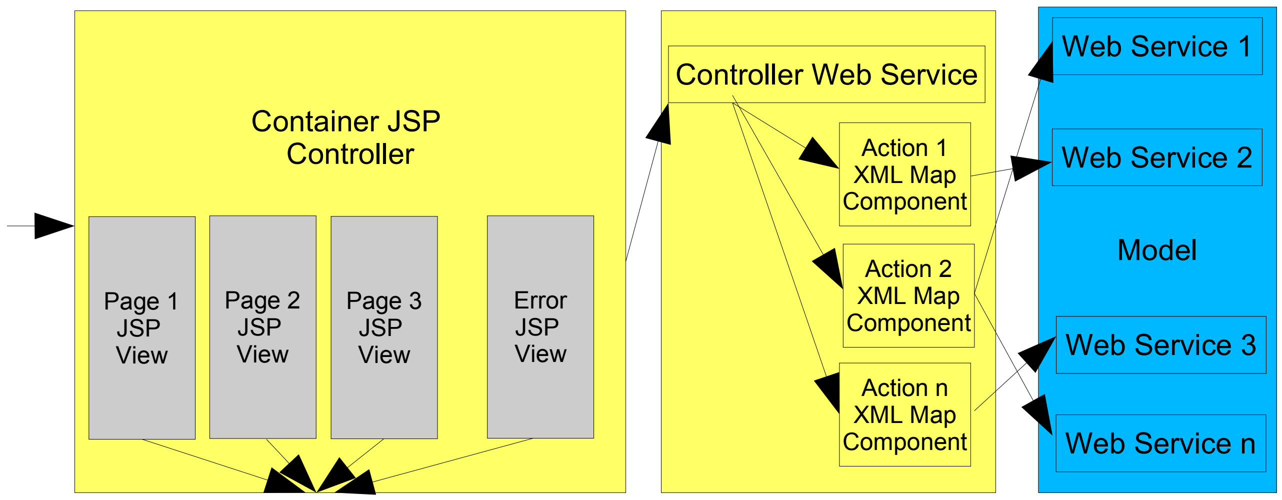
- the team has little or no Java/J2EE experience;
- the team does not posses licenses for exteNd Director (the preferred UI creation toolkit for Composer web services);
- the team is creating an HTML web application using mostly static or content managed pages and only a small part of the application needs to dynamically generate content.

Note: This document assumes that the reader already has a familiarity with Novell's exteNd Composer product.

# 2. Framework Structure

## *2.1 Control Flow*

The diagram below shows the basic structure of a ComposerMVC project. In a Composer project, the Container JSP and Web Service parts of the Controller are pre-supplied and do not need to be created or modified. The only part of the Controller requiring development effort is the creation of the individual Action components (XML Map components) which respond to HTML Form submissions.



**Key:** Blue = Model, Grey = View, Yellow = Controller

The typical flow of control for deployed ComposerMVC projects is:

1) Via a browser, an end user access the URL of the main controller JSP (Container JSP)
2) The main controller JSP determines that for this user, the first view should automatically be shown and forwards the request straight to the Page 1 JSP view to generate a HTML page.
3) The result HTML page typically contains an HTML Form, which includes two special hidden input items ('Controller' name and 'Action' name) in addition to normal HTML input items and submit buttons.
4) The user enters some data and presses submit, which sends the HTTP Post request back to the URL of the controller JSP.
5) The controller JSP uses the two hidden parameters ('Controller' name and 'Action' name) to determine which custom Action component to call and invokes the Action component passing the input DOM (which contains all the submitted input items).
6) The Action XML Map component converts the input DOM to the format required by the web service(s) it will call, validates that all correct input parameters for the web service are present, and then invokes the web service with the modified DOM.
7) Once the web service has completed, the Action component checks the result output DOM from the web service and then identifies which result view (JSP) should be shown as a result, passing this back to the calling Controller web service. The Action component also checks for any errors which may have occurred in the called web service. Any error messages generated are also placed in its output DOMs, for use by the Controller web service.

8) If the output DOMs received by the Controller web service from the Action component, contains the name of a View JSP to show, the Controller web services then forwards the request to the result view JSP along with the output DOM. If the Controller web service does not receive an output DOM containing the result view name, the Controller forwards the request with any error messages (if present) to the Error view JSP.

9) The result view JSP uses static HTML, normal JSP actions/tags and the Composer tag library to take the result DOM from the web service and render it in the result HTML page. If this page also contains an HTML Form and the user presses submit in this form, then the whole flow cycle is repeated again.

10)If the error pages is forwarded to, the error page displays a generic message and if any error messages are passed to it, displays theses as well (again using static HTML, normal JSP syntax and Composer tags).

## 2.2 Inputs and Outputs

Typically a JSP which contains a HTML Form for submission to a ComposerMVC application, will contain HTML source similar to the following:

```
<FORM METHOD="POST" ACTION='orders'>
      <INPUT TYPE="hidden" NAME="Controller" VALUE="ordersController">
      <INPUT TYPE="hidden" NAME="Action" VALUE="searchAction">

      <P>Surname: <INPUT TYPE="text" NAME="LastName"></P>
      <P>Account Id: <INPUT TYPE="text" NAME="Account"></P>

      <P><INPUT TYPE="submit" NAME="search" VALUE="Search"></P>
</FORM>
```

This example shows an order lookup form where the user is expected to enter his/her last name and account id to be able to lookup his/her existing orders.

In the ComposerMVC project, the main controller JSP will be associated with the web-application URL '/orders'. When invoked, the Controller JSP uses the standard Composer 'execute' tag to invoke the Controller Web Service which will receive this input DOM in the following format:

```
<ROOT>
  <Controller>ordersController</Controller>
  <Action>searchAction</Action>
  <LastName>Bloggs</LastName>
  <Account>8437393</Account>
  <submit>Search</submit>
</ROOT>
```

The controller web service is then able to invoke the Action Component (XML Map) called 'searchAction'. Using the standard Composer map instructions, this Action Component converts the input DOM into the DOM format required by the proper web service to be called. For example:

```
<AccountSearch>
  <LastName>Bloggs</LastName>
  <Account>8437393</Account>
```

```
        </AccountSearch>
```

The output of both an Action Component and the Controller Web Service is actually two DOMs: (i) a normal output DOM,  and (ii) a special 'metadata' DOM. The output DOM can contain anything and just reflects the output of the normal web service called from the model. The metadata DOM has a special structure used by the Controller Web Services and Action Components.

For this example the output DOM (originally from the normal model web service) may be similar to the following:

```
        <Account>
          <FirstName>Joe</FirstName >
          <LastName>Bloggs</LastName>
          <Addresses>
               <Address>
                       <House>32</House>
                       <Street>Sea Avenue</Street?
                       <Town>Sunnytown</Town>
                       <Postcode>ST2 1AZ</Postcode>
               </Address>
               <Address>
                       <House>ACME House</House>
                       <Street>ACME Park</Street?
                       <Town>Grey City</Town>
                       <Postcode>GC1 1BB</Postcode>
               </Address>
          </Addresses>
        </Account>
```

...and the metadata DOM may have the following content:

```
        <MetadataOutput>
          <TargetView>orders_showresults</TargetView>
          <Errors/>
        </MetadataOutput>
```

...however if there was a validation error or normal web service error, there may be an empty output DOM and the metadata DOM may have the following content:

```
        <MetadataOutput>
          <TargetView>orders_errorpage</TargetView>
          <Errors>
            <Message>An account last name has not been specified</Message>
            <Message>An account first name has not been specified</Message>
          </Errors>
        </MetadataOutput>
```

The 'TargetView' element of the metadata DOM tells the main controller JSP which normal view JSP should be forwarded to. The 'Errors' elements of the metadata data DOM is stores any error messages which need to be displayed to the user.

In the example where the validation error occurred, a separate error JSP is used to show the error messages. Alternatively, it is perfectly acceptable for the target view element to be set by the Action component the the name of original search JSP. In this case the search JSP can show the error messages on the same page as the offending input items, thus enabling the end user to re-submit with correct data.

For this example, if the validation was successful and the web service returned an Output DOM containing the account details and metadata DOM with the 'orders_results' JSP set as the target view, the 'orders_results' JSP could have the following example content to show the results:

```
<H1>Account Details</H1>
<P> First Name: <composer:value name="data" xpath="/Account/FirstName"/> </P>
<P> Last Name : <composer:value name="data" xpath="/Account/LastName"/>  </P>
<composer:forEach name="data" xpath="/Account/Addresses/Address">
      <P> Postcode: <composer:value xpath="Postcode"/>
</composer:forEach>
```

..and alternatively, if any errors were returned in the metadata, a special error JSP or even a normal error JSP could display the errors in a manner similar to the following:

```
<P>The input data was invalid, please try again.</P>
<composer:hasvalue name="data" part="Metadata" xpath="/MetadataOutput/Errors">
      <composer:forEach name="data" part="Metadata" xpath="/MetadataOutput/Errors">
            <P>Error: <composer:value xpath="Message"/> </P>
      </composer:forEach>
</composer:hasvalue>
```

Observe that in the three example JSPs (orders_search, orders_showresults and orders_errorpage) only standard HTML tags, standard JSP syntax and the Composer Standard Tag Library are used. Also the normal web services (in the model) take a normal single input DOM and return a normal single output DOM and are thus not directly tied into the ComposerMVC framework structure.

# 3. Starting a New Project

## *3.1 Steps*

To use the ComposerMVC framework in an exteNd Composer 5.0 project, either unzip and copy the skeleton 'composermvc' project or choose to create a new empty project and import the relevant Xobjects from the skeleton 'composermvc' project into the new project. The following list describes all the elements which need to be included in a project to incorporate the ComposerMVC framework:

1) Rename the main Container/Controller JSP, called 'myapp' to the name of the application it is required for (eg. 'orders').
2) Rename the main Controller Web Service, called 'myappController' to the same name as the controller JSP suffixed by the word 'Controller' (eg. 'ordersController').
3) Select the 'Tools | Project Settings' menu option in the Composer Designer and add two new project variables to specify the name of the first page and the error page, respectively, for the application. For example:
   - Name: 'ordersController_FirstView' – Value: 'orders_search'
   - Name: 'ordersController_ErrorView' – Value: 'orders_errorpage;
4) At the top of each View JSP include the following two lines to ensure that the Composer Tag Library is defined and the output and metadata DOMs from the main Controller web service are available to use in the JSP:
   ```
   <%@ taglib uri="/composer" prefix="composer" %>
   <% pageContext.setAttribute("data", request.getAttribute("data")); %>
   ```
5) In each View JSP that will contain an HTML Form, include two hidden parameters to enable the Controller to be used correctly when a form submit event occurs. These items should specify the name of the Controller web service and the name of the target Action component respectively. For example:
   ```
   <INPUT TYPE="hidden" NAME="Controller" VALUE="ordersController">
   <INPUT TYPE="hidden" NAME="Action" VALUE="searchAction">
   ```
6) An Action component will usually be required for each View JSP that contains an HTML Submit Form, to respond to the JSP's submit event. An Action component should have the following structure:
   - The Input DOM should use the 'MVC-ControllerInput' XML template.
   - The Output DOM can be free form (no template) or use the same output template that is used by the normal web service that will be called by this component.
   - An extra output DOM should be added and called 'Metadata' and this should use the 'MVC-MetadataOutput' XML template.
   - Create a single temporary DOM (eg. 'Temp') which can be free form (no template) or may use the same input template that is used by the normal web service that will be called.
   - Include validation logic (eg. Condition actions, Script, etc.) to validate that the input DOM content is correct and if not, set the target view and error message elements of the 'Metadata' output DOM accordingly (using map actions).
   - Map elements of the input DOM to the temporary DOM in the format which will be expected by the model web service to be called.

   - Use the Component Execute action to invoke the normal web service. Map the temporary DOM to the input of the service and map the output of the service straight to the Action component's output DOM.
   - Using a Map action, set the 'TargetView' element of the output 'Metadata' DOM to the literal name of the JSP to be shown next.

7) Ensure that all JSPs (both the Controller JSP and the View JSPs) are included in an application's Deployment profile with appropriate URLs specified

<u>Notes</u>

- The ComposerMVC framework provides a category of templates called 'MVC'. This contains the input template ('ControllerInput') and the metadata output ('MetadataOutput') template which are used by the Controller Web Service and each Action Component.
- The framework provides an ECMAScript library called 'ServletAPI'. This can be used to invoke various Servlet related APIs and is specifically used by the framework to temporarily store the previously returned web service output DOM on the user's session object.
- View JSPs can be arbitrarily named. However, experience has shown that prefixing the jsp name with the name of the application helps organise the project better. For example, for an 'orders' application, instead of calling the search JSP 'search', it should be called 'orders_search'.
- Action components can also  be arbitrarily named. However, experience has shown that suffixing the component name with the word 'Action'  helps to distinguish action components from the regular XML Map components in a project. For example, for an 'orders' application, instead of calling the search action component 'SearchComponent', it should be called 'searchAction'.
- The controller JSP and web service naming conventions specified in steps 1 and 2 above MUST be adhered to otherwise the framework will not be able to link the two together and the project variables listed in step 3 will not be located by the controller.
- Composer provides a set of Tag Library tags which can be executed from a JSP. These are described in the main Composer documentation (C:\Program Files\Novell\exteNd5\Docs\Start_Composer_Help.html -  Select 'Composer User's Guide' - Select 'The Composer Tag Library'). These tags include:
   - value
   - forEach
   - hasnovalue
   - hasvalue
   - if
   - execute
- To supplement the normal HTML tags, JSP tags and Composer tags, developers may also find that the JSP Standard Tag Library (JSTL – <u>http://java.sun.com/products/jsp/jstl</u>) helps in the authoring of View JSPs. To incorporate this Tag Library, just include the JSTL jar files in the Composer project and the appropriate 'taglib' directive at the top of each JSP which will use the JSTL tags.
- Instead of hard-coding the target URL for a HTML Form submit, the following JSP

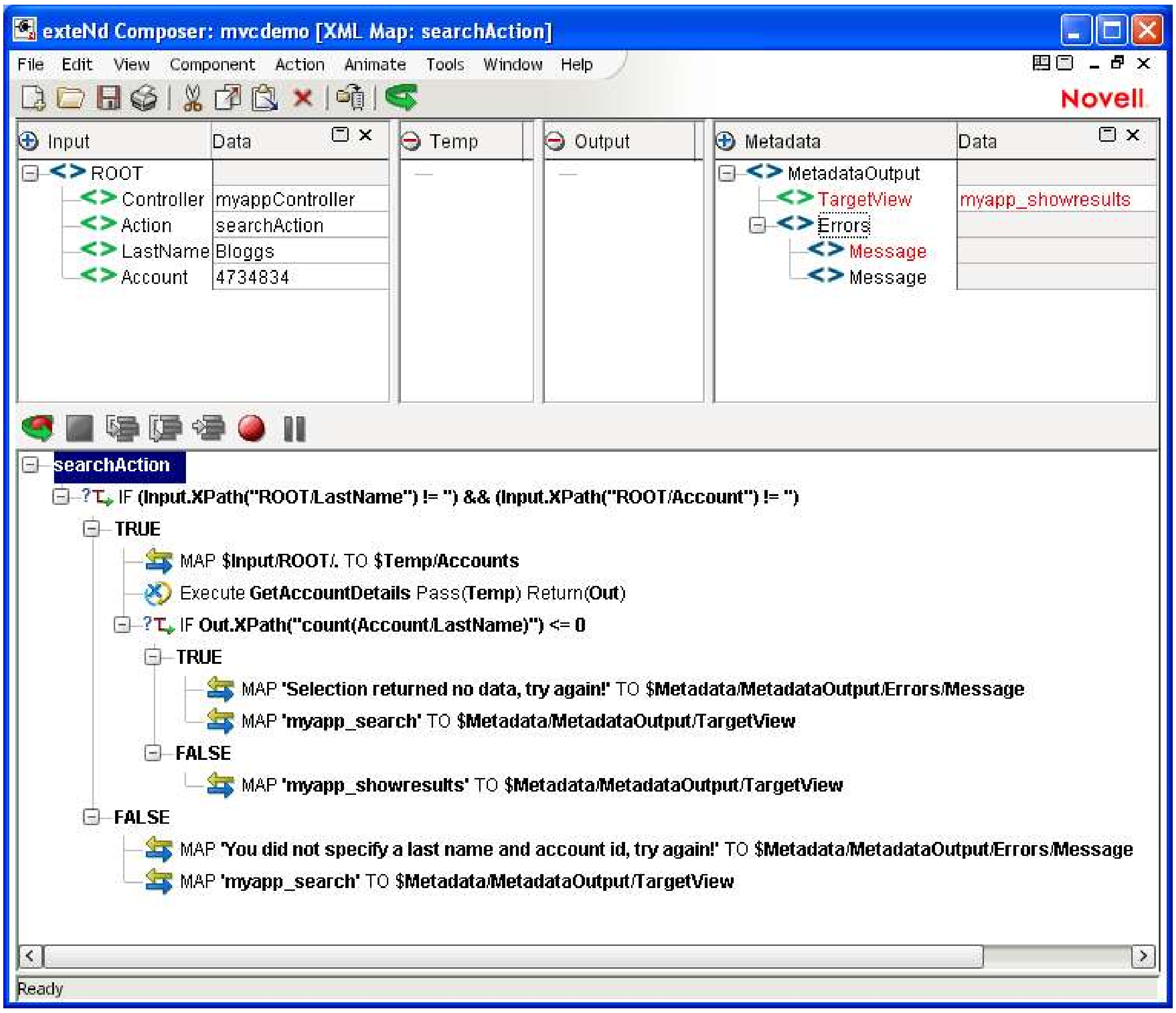
expression can be used to ensure that each View JSP always posts back to the main controller JSP:

```
<FORM METHOD="POST" ACTION='<%=request.getRequestURI()%>'>
```

- If required, view JSPs can still pull data directly from the model by using the Composer 'execute' tag to execute a named web service.
- <A HREF...> hyperlinks can also be generated in a page to call the controller JSP when selected, provided that the hyperlink URL parameters include the special 'Controller' and 'Action' parameters. The Controller Web Service will treat HTML Form POST submits and HTML hyperlink GET requests in the same way, assembling all the input parameters into an input DOM.

## *3.2 Example Action component*

Below is a screenshot of an example Action component called 'searchAction' which would be used to invoke an account search web service given a user's last name and account id.


exteNd Composer: mvcdemo [XML Map: searchAction]
File Edit View Component Action Animate Tools Window Help
Novell
Temp
Output
Ready

| Input | Data |
|---|---|
| ROOT | |
| Controller | myappController |
| Action | searchAction |
| LastName | Bloggs |
| Account | 4734834 |

| Metadata | Data |
|---|---|
| MetadataOutput | |
| TargetView | myapp_showresults |
| Errors | |
| Message | |
| Message | |

```
searchAction
  IF (Input.XPath("ROOT/LastName") != '') && (Input.XPath("ROOT/Account") != '')
    TRUE
      MAP $Input/ROOT/. TO $Temp/Accounts
      Execute GetAccountDetails Pass(Temp) Return(Out)
      IF Out.XPath("count(Account/LastName)") <= 0
        TRUE
          MAP 'Selection returned no data, try again!' TO $Metadata/MetadataOutput/Errors/Message
          MAP 'myapp_search' TO $Metadata/MetadataOutput/TargetView
        FALSE
          MAP 'myapp_showresults' TO $Metadata/MetadataOutput/TargetView
    FALSE
      MAP 'You did not specify a last name and account id, try again!' TO $Metadata/MetadataOutput/Errors/Message
      MAP 'myapp_search' TO $Metadata/MetadataOutput/TargetView
```

# 4. MVC Demo Application

A sample project called 'mvcdemo' is provided to demonstrate how the ComposerMVC project can be used in practice. This project can also be used as the basis of any new ComposerMVC project rather than using the skeleton 'composermvc' project, if preferred.

This project assumes that the full exteNd Enterprise suite is installed including MySQL and the 'samples50' database (the 'phonelist' table in this database is used by the project). The project provides a simple 3-stage page flow enabling a user to:

(i) search for a person by last name or email address

(ii) view the search results and select one person to view details for

(iii) view the selected persons full details

To open and run this project, unzip the project to the filesystem and open the project's SPF file using the Composer Designer.

Before the project can be run it may be necessary to change the database connection details for the 'samples50' MySQL database. From the Composer Designer, open the Connections – SamplesDB object and ensure that the JDBC URL, user and password fields are set correctly (the password is assumed to be 'admin' by default).

To run the project, select the Composer deploy option and either deploy the project to the default exteNd application Server profile or create and choose a new profile for another server (eg. Tomcat).

Once deployed, the application can be run from a browser via the URL 'mvcdemo/index.html' (eg. http://localhost/mvcdemo/index.html).

# A. Appendix

## A.1 Licence

All source code, binary releases and documentation which constitute the ComposerMVC project are covered by the following open source BSD-style licence: