NEF Sampler Application

Tutorial

Palmer Business Park
4550 Forbes Blvd, Suite 320
Lanham, MD 20706

(301) 879-3321

http://www.netspective.com
info@netspective.com

Copyright

Disclaimer and limitation of liability

Customer Support

Customer support is available through e-mail via support@netspective.com

NEF Sampler Application Tutorial

Table of Contents

1. Objectives

Welcome to the NEFS Sampler Tutorial (Tour)! This tutorial aims at giving you a tour of the NEFS Sampler. It facilitates your learning by reducing the learning time through quick familiarization with various Sampler sections. You don't have to worry about learning the use of Sampler. Instead, you can concentrate on the core concepts being highlighted by the Sampler sections.

1.1. Learning Objectives Checklist

At the end of this tutorial, you should be able to understand:

- the objectives of NEFS Sampler
- how to set up the NEFS Sampler at your application server
- how to view the XDM code within NEFS Sampler
- purpose of NEFS Sampler sections
- usage of NEFS Sampler sections

2. Assumptions

For the purpose of this tutorial, we will be assuming that you installed the following:

- Java Developer's Kit (JDK) 1.2, 1.3 or 1.4 See Section 2.1, “Downloading Java Developer's Kit (JDK)”
- An Application Server (servlet container) supporting the Servlet 2.2 or higher specification See Section 2.2, “Application Server (Servlet Container)”

We assume that you are using the default port `8080` for your web server. If you chose different values for the installation path and the port number, you should substitute the paths and the URLs in our example with your values as needed. This tutorial also assumes your familiarity with XML, SQL, Java, Servlets and JDBC.

2.1. Downloading Java Developer's Kit (JDK)

Since NEFS comprises Java libraries, a fundamental requirement to develop applications with it is a Java SDK (the full SDK is required, the JRE will not be enough). You can obtain Sun’s official Java SDK from its Java web site at http://java.sun.com/j2se/1.4/download.html. This is a link to the Java 1.4 SDK but Java 1.2 and 1.3 will also work.

2.2. Application Server (Servlet Container)

Since Sparx works with standard J2EE application servers, a Servlet container is required if you're going to use Sparx. Both Axiom and Commons work in web-based or non-web-based applications but Sparx is a web application development library so an application server with a Servlet 2.2 or better container is necessary. Sparx-based applications have been tested on the following application servers:

- Apache Tomcat[2] (free)
- Caucho Resin[3] (free for development, commercial license required for deployment)
- BEA WebLogic[4] (commercial)
- IBM WebSphere[5] (commercial)

[2] http://jakarta.apache.org/tomcat
[3] http://www.caucho.com
[4] http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/server

- ORACLE Application Server[6] (commercial)
- Macromedia JRun[7] (commercial)

Note

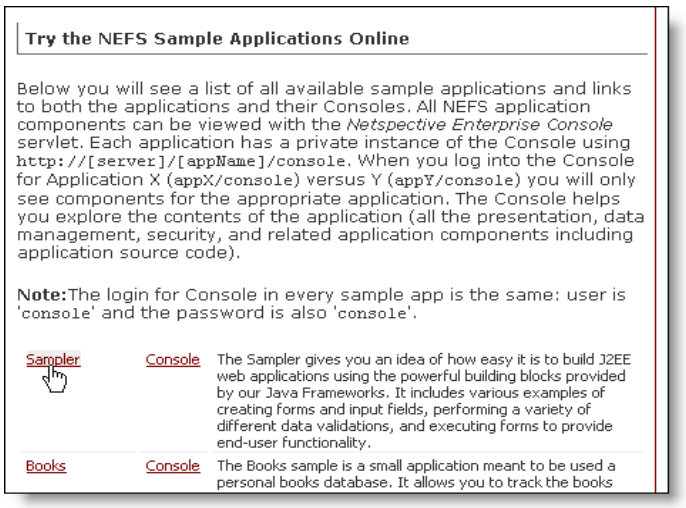
We recommend the Caucho Resin[8] application server if you're not familiar with other Servlet containers or if you're new to Java/J2EE application servers. It's an easy to install, easy to use, and fast Servlet container with advanced features that rival other more expensive application servers such as WebLogic and WebSphere. Resin is free for development use but requires a paid license before putting your application into production use. Rest assured though that all Sparx-based applications you write, even on Resin, will remain app-server neutral.

3. Starting the NEFS Sampler

There are two ways of exploring the NEFS Sampler:

3.1. Trying the NEFS Sampler Application Online

You may test the Sampler Application online at http://www.netspective.com/corp/products/frameworks/try All you need is a web browser and you can start using the NEF Sampler immediately.



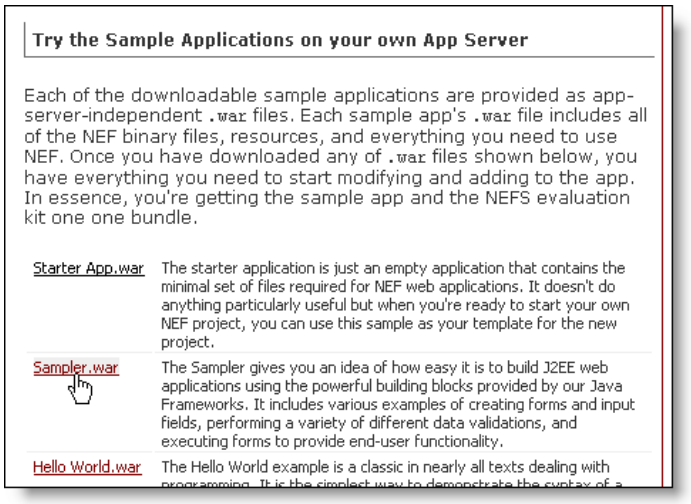
3.2. Setting up the NEFS Sampler Application on Your Own App Server

You may download the NEFS Sampler application as app-server-independent .war file. The Sampler Application's .war file includes all of the NEF binary files, resources and everything you need to use NEF. Once you have downloaded the .war file, you have everything you need to start modifying and adding to the app.

3.2.1. Downloading the Sampler Application

You can download the NEF Sampler Application file from http://www.netspective.com/corp/downloads/frameworks/samples

[5] http://www.ibm.com/websphere
[6] http://www.oracle.com/appserver/
[7] http://www.macromedia.com/software/jrun/
[8] http://www.caucho.com

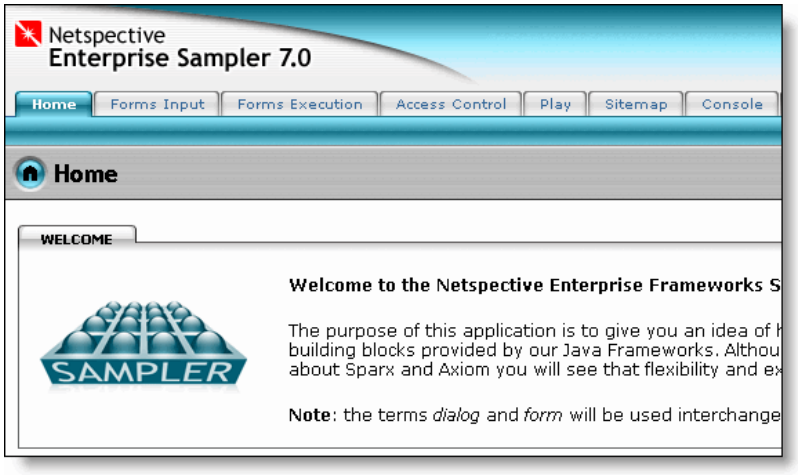


3.2.2. Setting up the Sampler Application

Copy the downloaded `nefs-sampler.war` file to the `webapps` folder of your servlet container (application server) and run (or restart) the application server. This will create an Application Directory Structure containing the necessary NEF files and sub-folders, under the `webapps` folder of your application server.

3.2.3. Testing the Sampler Application in a Browser

Use a web browser to access the root of the Sampler Application using the URL of the form http://*host*:*port*/nefs-sampler. If everything worked as it should, you will see the Sampler Application Welcome Page.



4. NEFS Sampler Objectives

As the name suggests, the purpose of NEFS Sampler Application is to give you an idea of how easy it is to build J2EE web applications using the powerful building blocks provided by our Java Frameworks. The NEFS Sampler uses an interactive mode thus making it easy to understand and use the samples. It includes various examples of creating forms and input fields, performing a variety of different data validations and executing forms to provide end-user functionality.

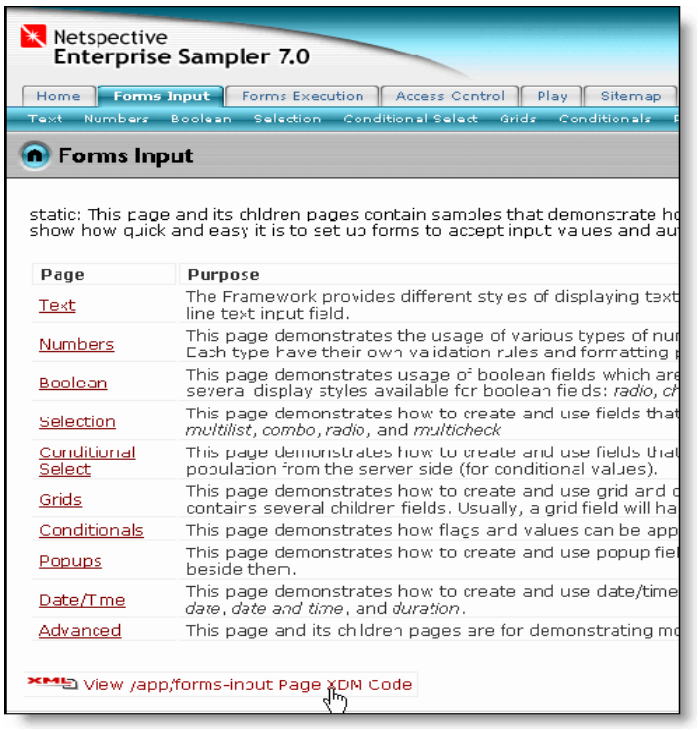
Although you start with simple NEFS usage examples, as you learn more about Sparx and Axiom you will see that flexibility and extensibility are also a part of their attributes.

5. Viewing the XDM Code

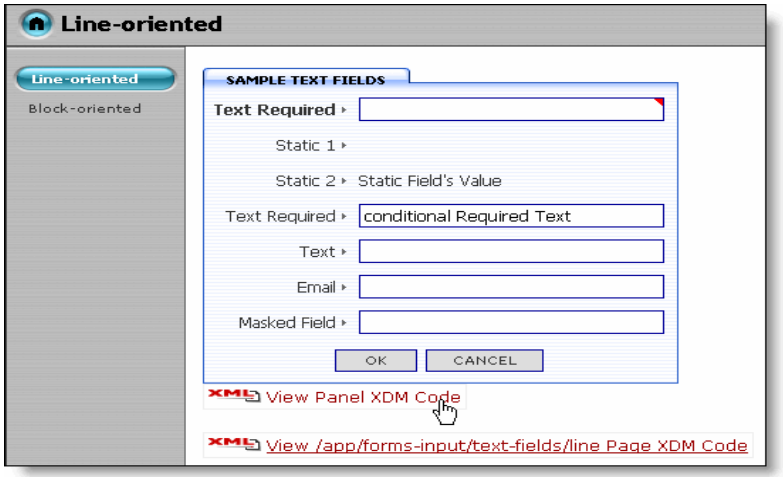
The eXtensible Markup Language (XML) plays an important role in NEF's ease of use, extensibility, and code generation. NEF declarations are performed using XML -- all dialogs, fields, validation rules, some conditional processing, all SQL statements, dynamic queries, configuration files, database schemas, and many other resources are stored in XML files that are re-usable across applications.

XDM is an acronym for "XML Data Model" and is designed to help Java programmers construct and configure Java objects using XML files without worrying about parsing and error checking. The `APP_ROOT/WEB-INF/sparx/project.xml` file uses XDM to declare all project components.

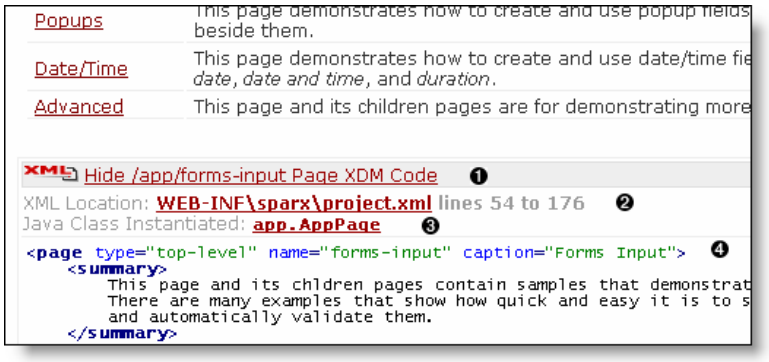
In the Sampler app if you see the XML icon along with a *View Page XDM Code* link, at the bottom of the page, you can view the Netspective Xml Data Model (XDM) source code for various objects on the page. This helps you view the XML code and see it working on the same page.



The XML icon also appears under various panels (like forms/dialogs) to see their code as well.

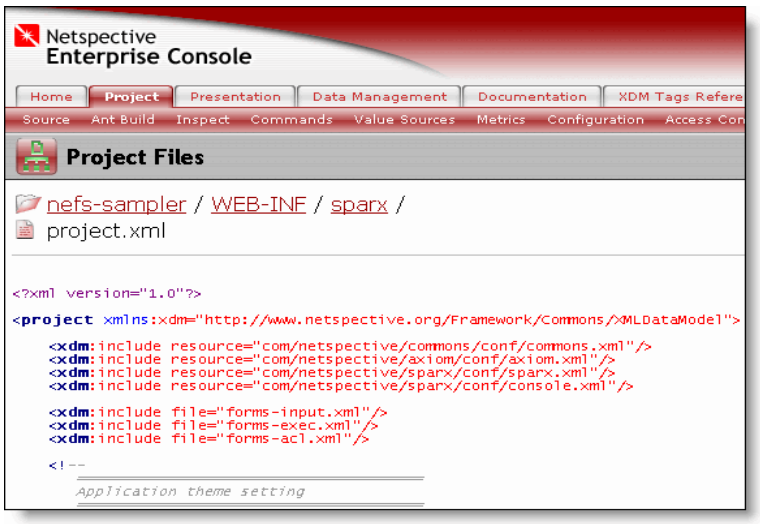


Clicking on the *View Page XDM Code* link to see the XML code (as shown below):



1. The link to hide the currently visible XDM code. Clicking on this link hides the currently visible XDM code.

2. Location of the currently open XDM code in the application's project file (`WEB-INF/sparx/project.xml`). Clicking on this link takes you to the Project | Project Files section of the Sampler app's Console, with Sampler app project file being displayed in browse mode.

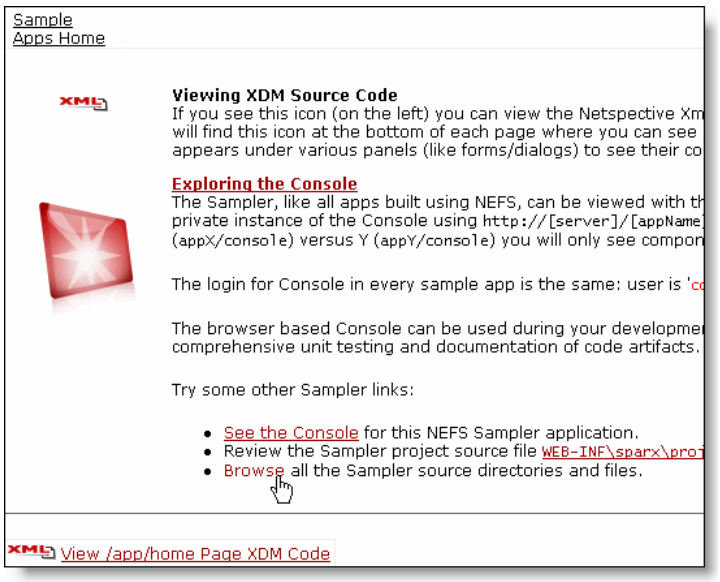


3. Java class file being instantiated for this page. Clicking on this link takes you to the Project | Project Files section of the Sampler app's Console, with the selected java file (`AppPage.java` in this case) being displayed in browse mode.

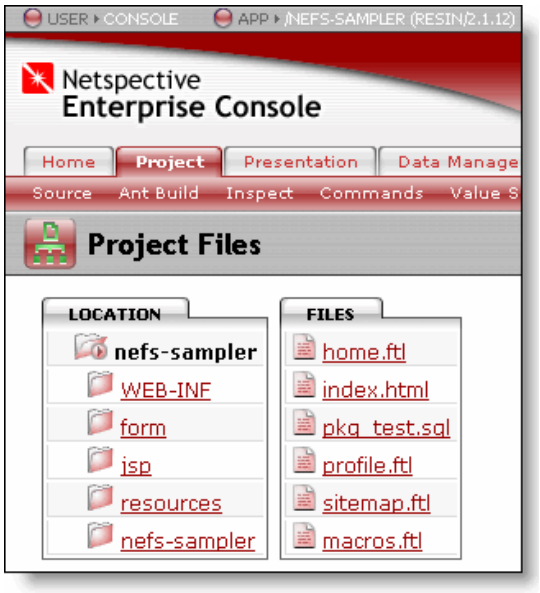
4. XDM code (from the Sampler app project file) corresponding to the currently opened page.

6. Viewing Project Folder and Files

At the end of the Sampler App Home page, there is a link for browing through all the source directories and files of the Sampler App.



When you click on the *browse* link, it takes you to the Project | Project Files section of Sampler app Console, displaying all the source folders and files for the Sampler app.



7. Sparx Navigation Tree Overview

The NEFS presentation layer tags are broken up into several groups: navigation, forms/fields, on-screen validation, and conditional processing. Almost all of the presentation layer tags are managed by Sparx.

7.1. Introduction to Navigation Declaration Tags

The navigation tags comprise primarily the `navigation-tree`, `page`, and `body` tags. These are very high-

level tags that provide a great deal of functionality without requiring any HTML or JSP. However, if you need to customize the behavior you have full access to the APIs through both inheritance and delegation. You focus on the hierarchy, specify the structure and rules and NEF will do the rest. The Sparx navigation system fully implements MVC (model-view-controller) design pattern.

Example 1. Navigation Tags in the Presentation Layer

```
<project xmlns:xdm="http://www.netspective.org/Framework/Commons/XMLDataModel">
...

<navigation-tree name="app" default="yes"> ❶

    <page name="home" default="yes" caption="Hello" heading="Hello World!"> ❷
      <body> ❸
        <![CDATA[


   This is the 'Hello World' app. Click <a href="next-steps">here</a> to see what's

    next.]]>
        </body>
    </page>


    <page name="next-steps" caption="Next Steps" heading="What's next?">
        <body source="next-steps.ftl"/> ❹
    </page>

    <page name="some-stuff" caption="Panels instead of Body">
        <panels> ❺
            <panel type="command" command="query,org.get-sponsor-info-by-id,-,-,-
,detail-compressed"/>
            <panel type="command" command="query,org.get-org-addresses-by-id,-,-,-
,report-compressed" />
        </panels>
    </page>

    <page name="sampler" caption="Sampler" redirect="vs-expr:${netspective-
url:sampler}"/> ❻
    <page name="sampler" caption="Sampler" include="/some/file.jsp"/> ❼
    <page name="sampler" caption="Sampler" forward="/some/file.jsp"/> ❽

    <page name="add" caption="Add Book" command="dialog,schema.db.Book Info,add"/> ❾
    <page name="edit" caption="Edit Book" command="dialog,schema.db.Book_Info,edit"
          require-request-param="id" retain-params="id">
        <missing-params-body> ❿
           Please choose a book to edit from the &lt;a href='home'&gt; books

           list&lt;/a&gt;.
        </missing-params-body>
    </page>
</navigation-tree>
```

❶ The `navigation-tree` tag starts out the definition of a tree. You may declare as many navigation trees as your application needs. Consider different trees for different users (based on personalization) or a different tree for each access-control role (for security) or any other criteria required by your application. Each tree has a name and may be marked with `default=yes` if it is to be the default tree. Which tree is actually used by the application may be specfied as a servlet parameter or chosen dynamically at runtime based on some processing rules.

❷ Each `navigation-tree` tag one or more `page` tags that define what pages should be visible to the user. Each `page` tag supports common things like the page's name (unique identifier), a caption (what might show up in a tab or menu), a heading, and title.

❸ Each `page` tag may have an optional `body` tag with contents directly specified as XML content. You can use the CDATA XML element to escape HTML tags. The content of the `body` tag is treated as a FreeMarker[11] template by default.

[11] http://www.freemarker.org

❹ The `page` tag's `body` tag may instead specify a source attribute that indicates the given file is a a FreeMarker[12] template relative to the `APP_ROOT` directory. Other template engines like JSP are also supported.
❺ The `page` tag's `body` tag may be replaced with the `panels` tag to have automatic layouts of mutliple panels.
❻ The `page` tag may specify a redirect attribute to automatically redirect to another page whenever it is chosen.
❼ The `page` tag may specify an include attribute to simply insert the contents of another web resource directly into the body.
❽ The `page` tag may specify a forward attribute to forward the request to another web resource (this is the same as Servlet forwarding not HTTP forwarding).
❾ The `page` tag may specify a command attribute to use the Command pattern and delegate the body to a `com.netspective.commons.command.Command` interface. These are very high-level commands like interacting with a dialog and displaying the results of a query that can be defined elsewhere and called on different pages.
❿ The `page` tag may specify required request parameters and optionally produce automatic error messages when they are not provided.

Note

Almost all `page` tag attributes support *Value Sources* for their values so the values may be static or dynamic. This allows, for example the captions, headings, and all user-visible values to be properly placed in property files or other external resources and mapped in a language-specific manner for internationlization purposes. The important thing to understand is that even though the declarations are in a static XML file, the values of the descriptors are capable of being dynamic.

Note

Please see the NEF User's Manual[13] for instructions on how to use the Console to review tag documentation and use the Tags Tree to get further details on the `navigation-tree` and `page` tags.

8. NEFS Sampler Main Components

The NEFS Sampler groups the samples into following logical sections:

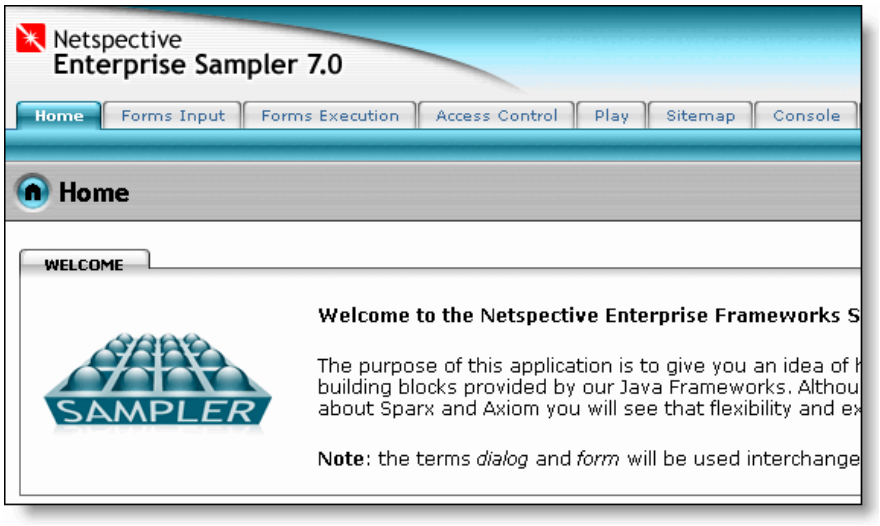
- Home
- Forms Input
- Forms Execution
- Access Control
- Play
- Sitemap
- Console
- Sample Apps Home

8.1. Home

This is the Welcome page for the Sampler. It outlines the objectives of the Sampler and also contains a list of links pointing to the logical sections of the Sampler. You may access the Sampler sections by these links or the

[12] http://www.freemarker.org
[13] http://www.netspective.com/corp/resources/support/docs/nef-manual/index.html

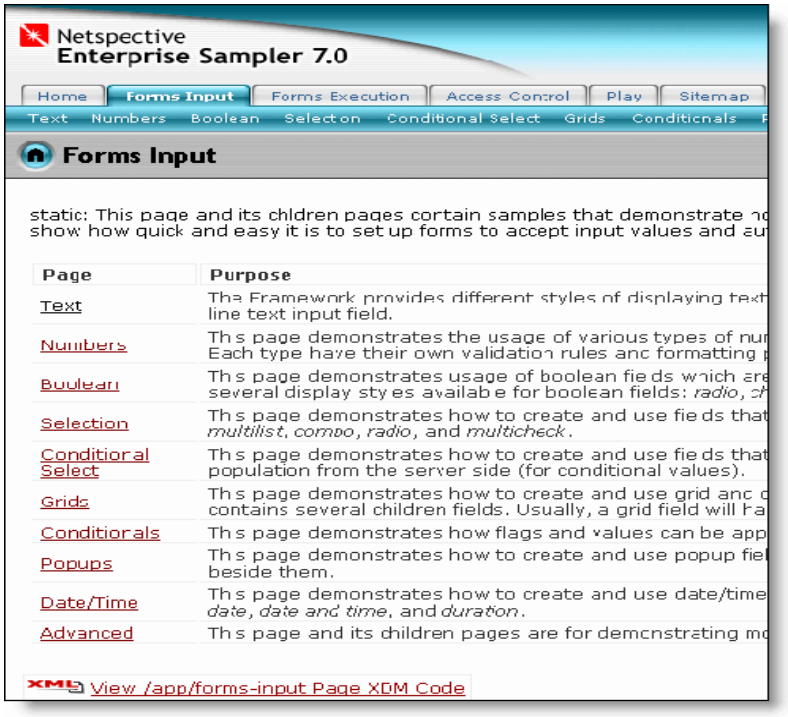
tabs provided the top of the page.



8.2. Forms Input

This module contains samples that demonstrate how forms and input fields are created in Sparx. There are many examples that show how quick and easy it is to set up forms to accept input values and automatically validate them.

The examples contained in this section are further divided into sub-categories. The *Forms Input* page contains a list of links to its children pages containing the categorized example pages. Links to these children pages are also provided through a submenu bar under the *Forms Input* tab.



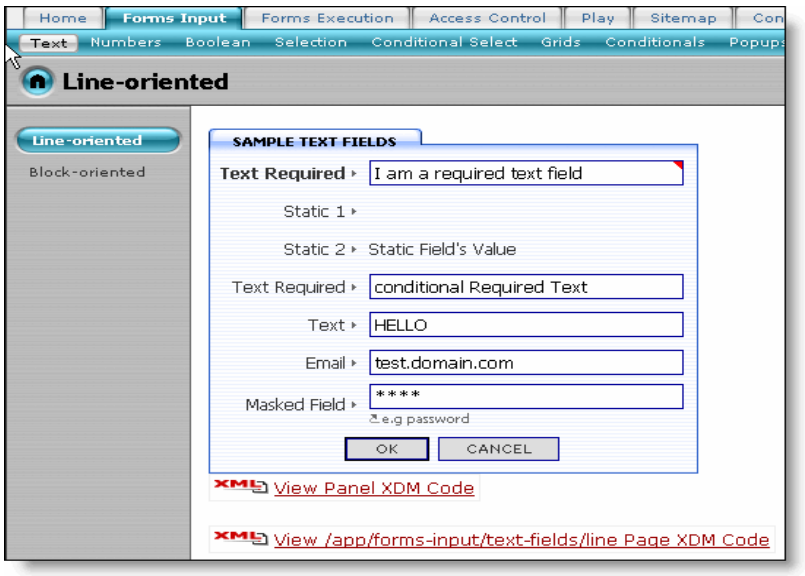
8.2.1. Text

The Framework provides different styles of displaying text input fields. One is a simple one line text input field

while another is a multi-line text input field.

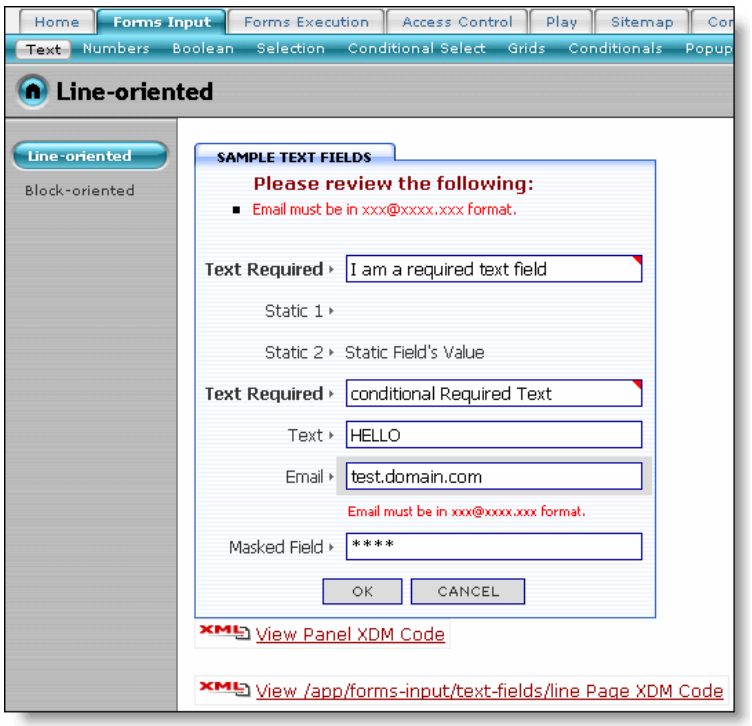
The Line-based text input examples are included under Line-oriented submenu.

Figure 1. Sample Page for Line-oriented Text Fields



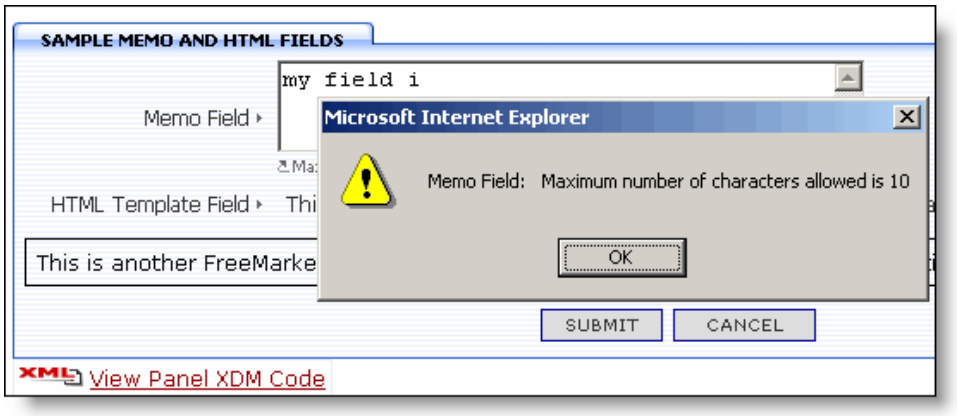
The sample text fields on this page show the usage of static text fields, input text fields (text boxes), email fields and masked fields for password entry. You can also see how to associate hints with a particular text input field. The example also implements field validation. Click on the **OK** button to see how field validation messages are displayed.

Figure 2. Field Validation Messages



At times you need multi-line fields (e.g. text area) for your text input. The Block-oriented submenu contains examples highlighting the block based text input.

Figure 3. Sample Page for Block-oriented Text Fields

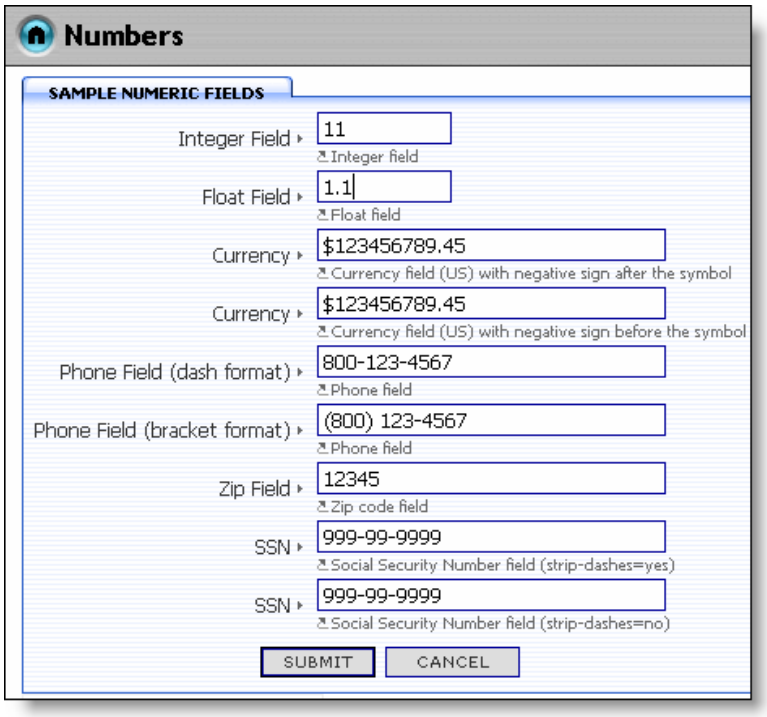


This page shows the usage of memo and template fields using custom CSS and HTML. The sample page also incorporates error messages using message boxes (as shown in the above image).

8.2.2. Numbers

This page demonstrates the usage of various types of numeric fields that are available: float, integer, currency, phone, and zip code. Each type have their own validation rules and formatting patterns.

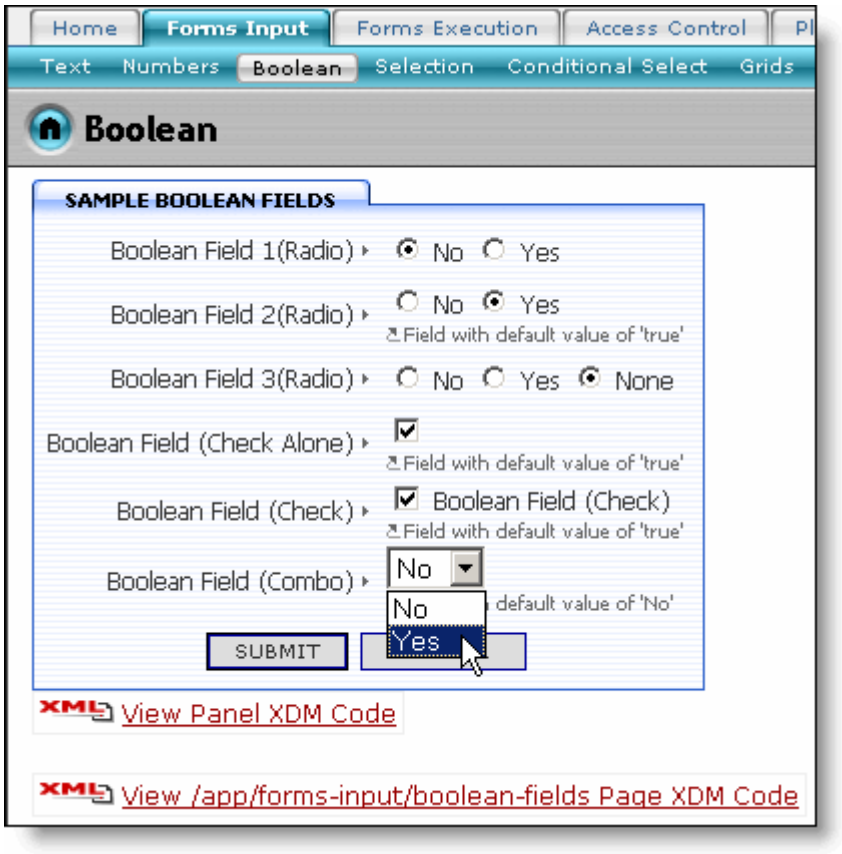
Figure 4. Sample Page for Numeric Fields



8.2.3. Boolean

This page demonstrates usage of boolean fields which are fields that can only have a value that is either true or false. There are several display styles available for boolean fields: radio, check-alone, popup, and combo.

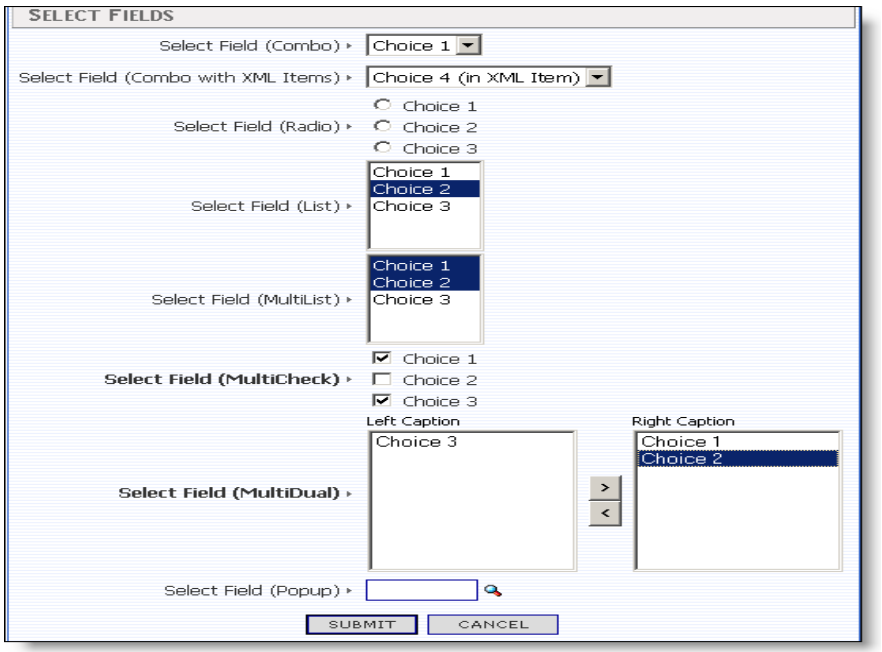
Figure 5. Sample Page for Boolean Fields



8.2.4. Selection

This page demonstrates how to create and use fields that allow selection of values. There are several styles available: list, multidual, multilist, combo, radio, and multicheck. You can also see the implementation for popup based select field.

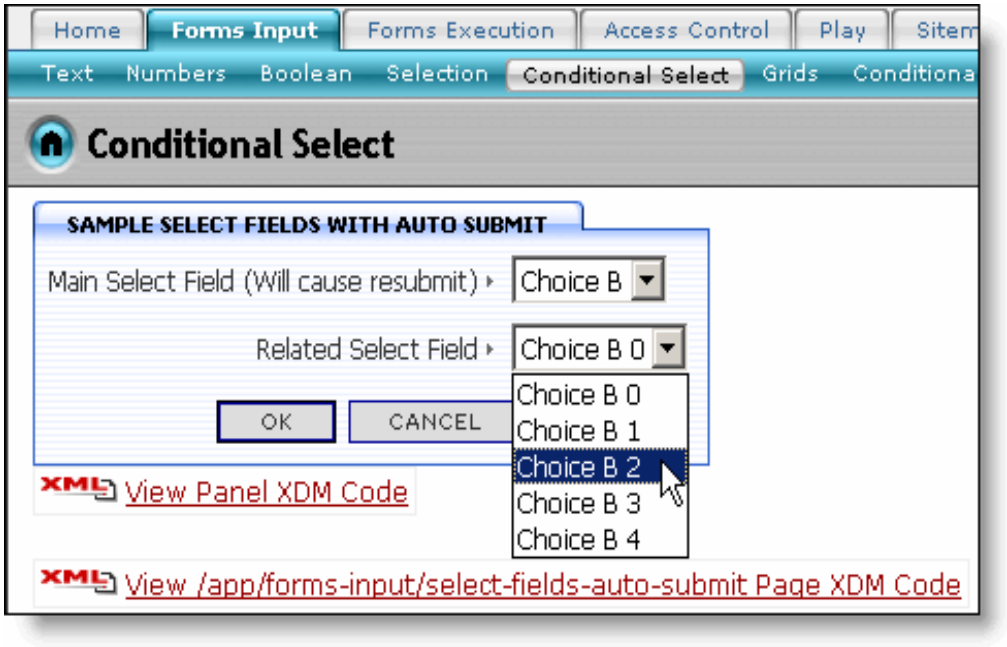
Figure 6. Sample Page for Select Fields



8.2.5. Conditional Select

This page demonstrates how to create and use fields that allow selection of values and auto submit forms that will do other data population from the server side (for conditional values).

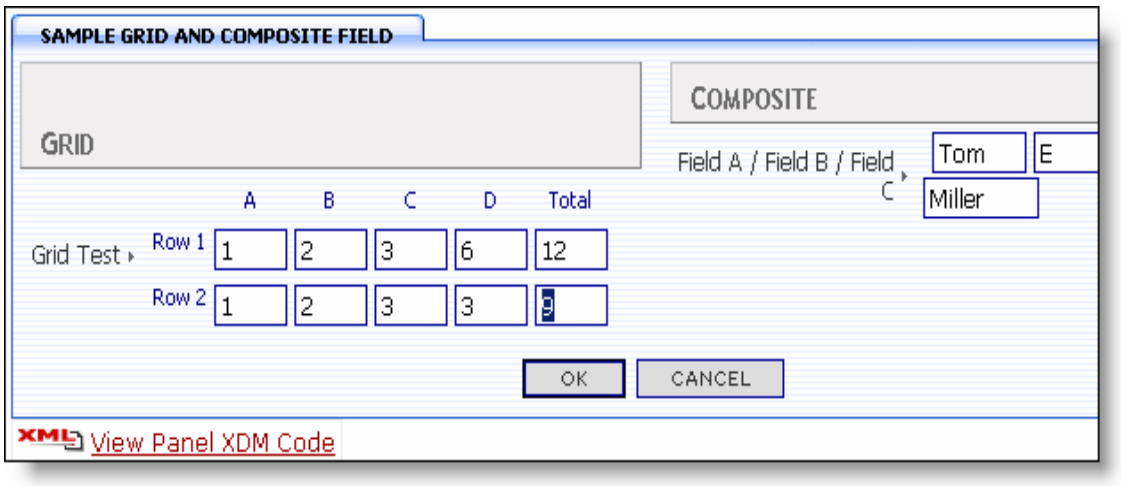
Figure 7. Sample Page for Conditional Select Fields



8.2.6. Grids

This page demonstrates how to create and use grid and composite fields. A composite field can be described as a logical field that contains several children fields. Usually, a grid field will have multiple composite fields which act as rows.

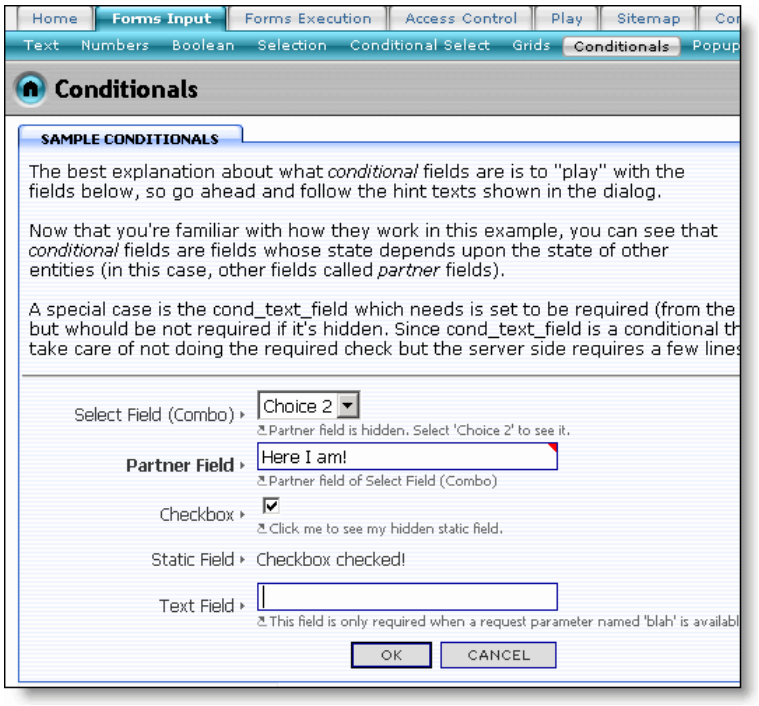
Figure 8. Sample Page for Grid and CompositeFields



8.2.7. Conditionals

This page demonstrates how flags and values can be applied to a dialog field based on conditionals defined for that field. Conditional fields are fields whose state depends upon the state of other entities (in this case, other fields called *partner* fields).

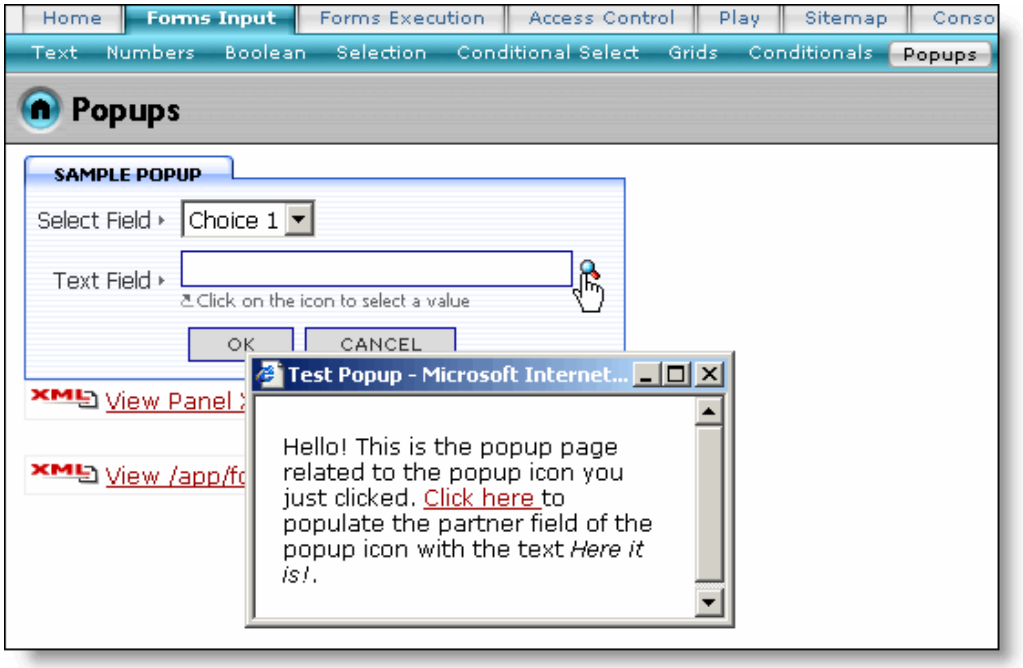
Figure 9. Sample Page for Conditional Fields



8.2.8. Popups

This page demonstrates how to create and use popup fields. Sparx provides a feature to display a popup dialog box when a popup icon is clicked and values from a selection in the popup can be returned to the main window. Popup fields are visually identifiable by their special magnifying glass icon beside them.

Figure 10. Sample Page for Popup Fields

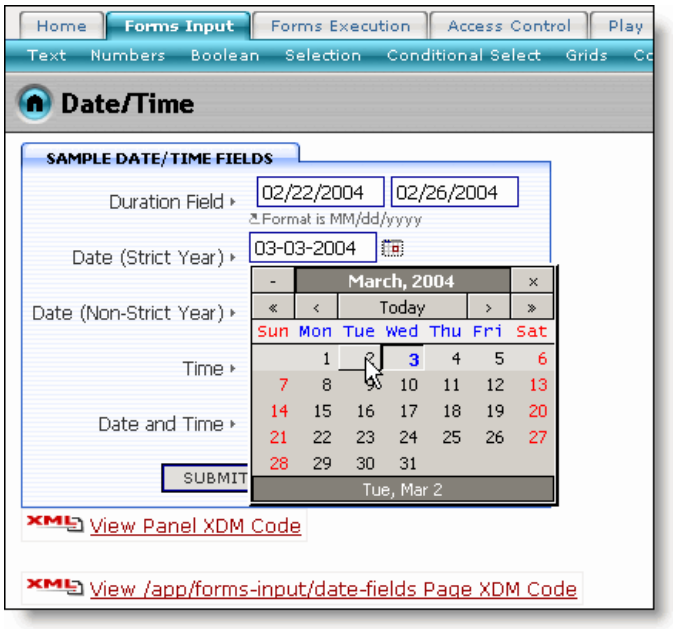


8.2.9. Date/Time

This page demonstrates how to create and use date/time fields. There are several versions of date/time fields that

are available: time, date, date and time, and duration.

Figure 11. Sample Page for Date/TimeFields



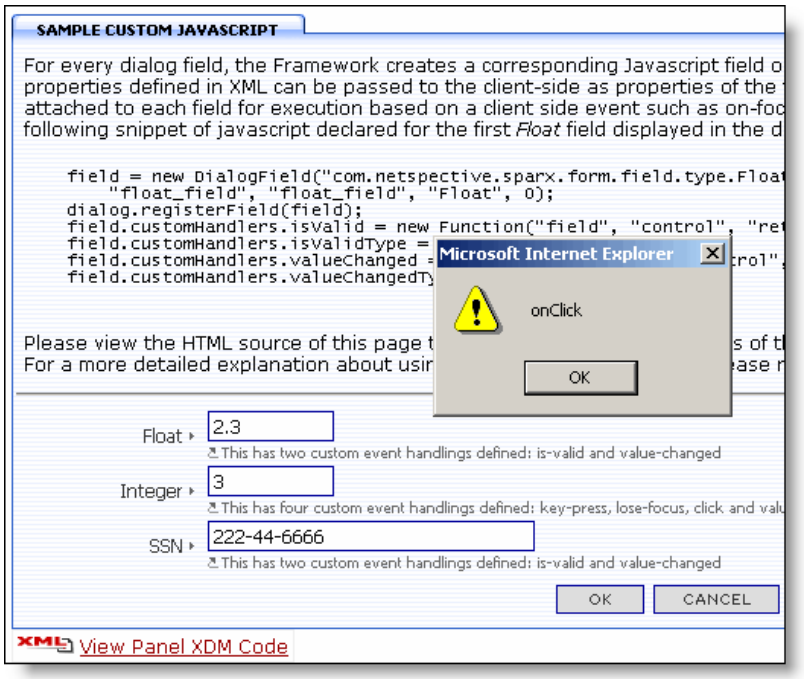
8.2.10. Advanced

This page and its children pages are for demonstrating more advanced usage of the built-in features of the dialog fields.

8.2.10.1. Client-side Script

This page demonstrates how to "plug" *custom javascript* calls to the client-side events (e.g. click, value-changed, lose-focus, key-press, is-valid) of the dialog fields.

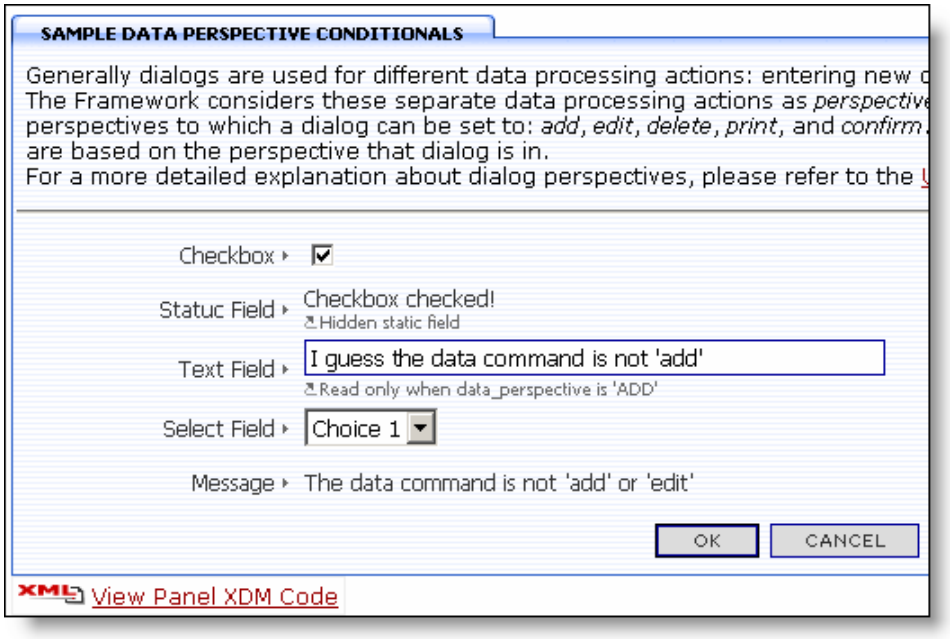
Figure 12. Sample Page for Custom Javascript



8.2.10.2. Perspectives

Generally dialogs are used for different data processing actions: entering new data, editing existing data, and deleting existing data. The Framework considers these separate data processing actions as *perspectives* of the dialog. Thus, there are several different perspectives to which a dialog can be set to: *add*, *edit*, *delete*, *print*, and *confirm*. This page demonstrates the usage of data perspectives for filling dialog fields.

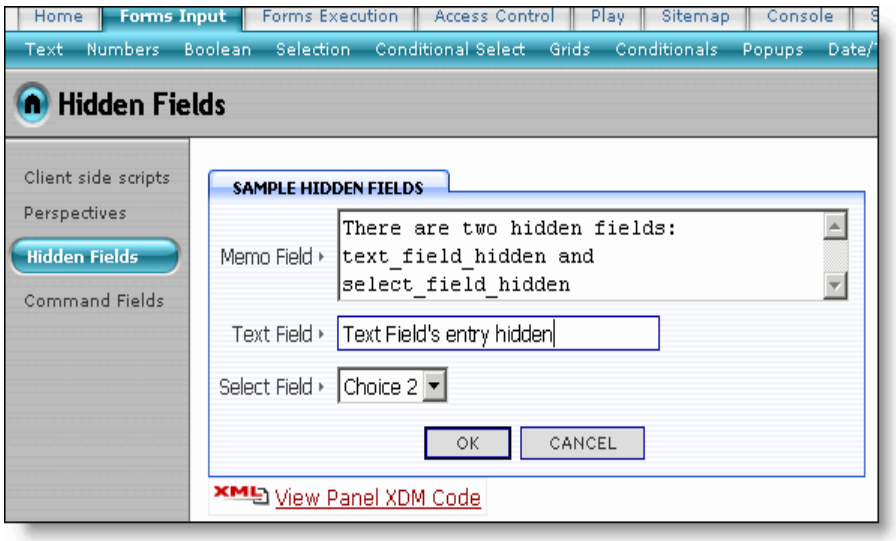
Figure 13. Sample Page for Perspectives



8.2.10.3. Hidden Fields

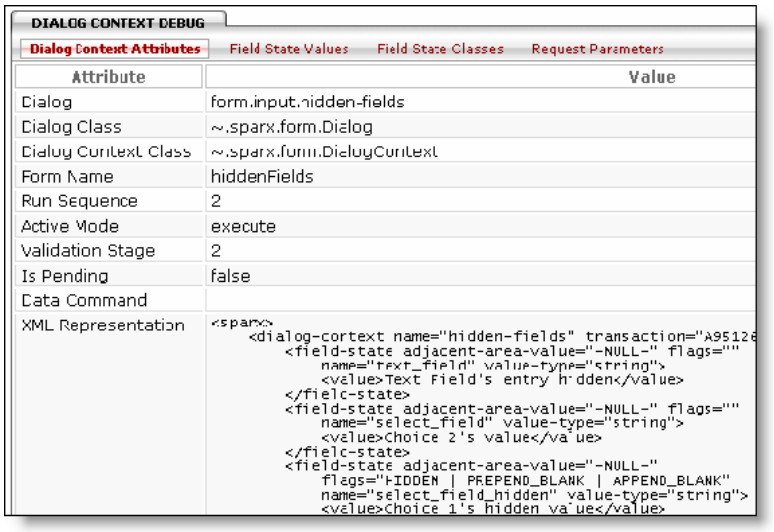
This page demonstrates different usages of *hidden fields* in a dialog. Hidden fields are useful for keeping track of information that you don't want the user to see.

Figure 14. Sample Page for Hidden Fields



Click the **OK** button. The dialog context is displayed in debug mode (as shown below). You may check the values of the hidden fields and other dialog fields from there.

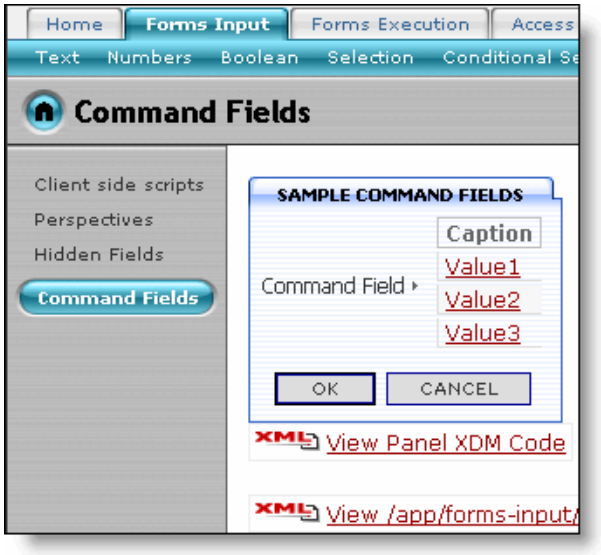
Figure 15. Dialog Context Debug Information



8.2.10.4. Command Fields

This page demonstrates the usage of a `List` command to populate a dialog field. Only certain commands can be used for dialog fields. Commands execute arbitrary tasks defined either by you or the framework. They are used to encapsulate common logic and reuse that logic across pages and dialogs (forms). For information on *Commands* in general, please consult the *User Manual*.

Figure 16. Sample Page for List Command



8.3. Forms Execution

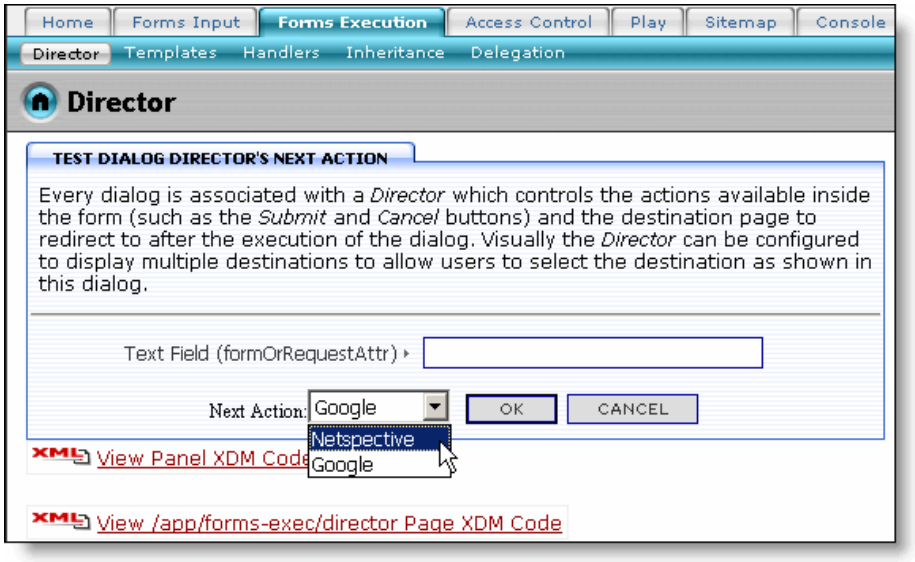
This module contains samples that demonstrate how forms are executed in Sparx. While the Forms Input module described how to create forms and fields, this module deals with how to perform actions based on already validated input. Numerous examples delve into how to perform custom validation and execute dialog handlers based on existing Sparx functionality or creating your own through inheritance and delegation.

The examples contained in this section are further divided into sub-categories. The *Forms Execution* page contains a list of links to its children pages containing the categorized example pages. Links to these children pages are also provided through a submenu bar under the *Forms Execution* tab.

8.3.1. Director

This page demonstrates the usage of a *Dialog Director* which controls destinations after successful form submission and the display of the dialog buttons.

Figure 17. Sample Page for Director



Select the value from the `Next Action` select box and click the **OK** button to see the execution of next action for this dialog.

The XML Code that configures the Dialog Director is given below:

Example 2. XML for Using the Dialog Director

```
<project>
...
 <dialog type="exec-sample" name="director-next-action"
         redirect-after-execute="yes"> ❶
  <frame heading="Test Dialog Director's Next Action"/>
   <field type="html">
    <body>
      <![CDATA[
        <div class="textbox">
         Every dialog is associated with a <i>Director</i> which controls the
         actions available inside<br>
         the form (such as the <i>Submit</i> and <i>Cancel</i> buttons) and the
         destination page to<br>
         redirect to after the execution of the dialog. Visually the <i>Director</i>
         can be configured<br>
         to display multiple destinations to allow users to select the destination
         as shown in<br>
         this dialog.
        </div>
        <hr/>
      ]]>
    </body>
   </field>
   <field type="text" name="text field" caption="Text Field (formOrRequestAttr)"
        default="request:value"/>

   <director> ❷
     <next-actions caption="Next Action\:"
                   choices="text-list:Netspective=http\://www.netspective.com;
                                      Google=http\://www.google.com"
                   display-single-action="yes"/> ❸
   </director>
</dialog>
...
</project>
```

❶ The `dialog` tag defines a dialog using `exec-sample` template as the dialog-type.
❷ The `director` tag is used to define the director to be associated with this dialog.
❸ Configuring the director to display multiple destinations.

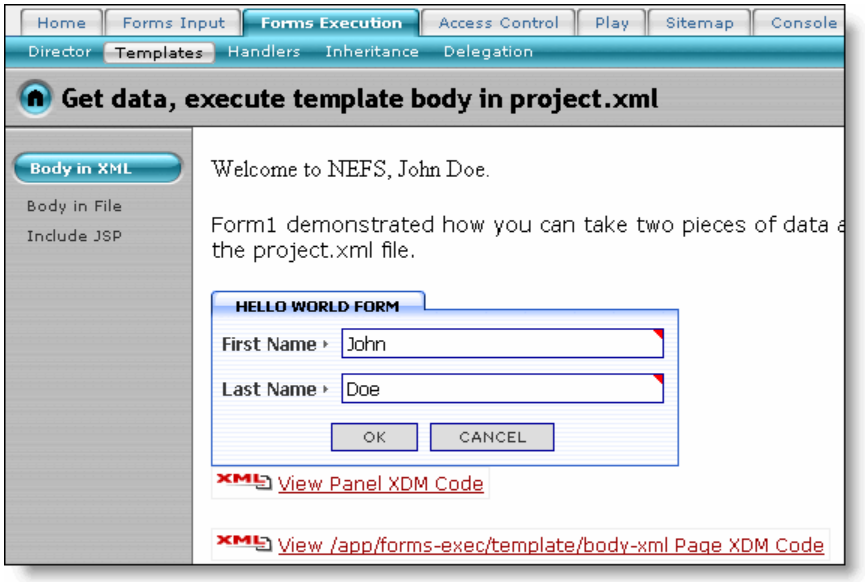
8.3.2. Templates

This page demonstrates the ability to *execute* templates or display JSP output after successful submission of a dialog. Usually executing a template means displaying the output from a template processing engine.

8.3.2.1. Body in XML

This page contains an example of a dialog which will execute a template body defined in the dialog's XML declaration and display the output. The XML template is used to display the Welcome message to user.

Figure 18. Sample Page for Templates using Body in XML



The XML Code for using the FreeMarker™ template (embedded within `project.xml`) is given below:

Example 3. XML Code for Using Template (embedded in project.xml)

```
<project>
...
 <dialog type="exec-sample" name="body-xml"> ❶
    <frame heading="Hello World Form"/>
    <field type="text" name="first_name" caption="First Name" required="yes"/>
    <field type="text" name="last name" caption="Last Name" required="yes"/>


    <on-execute handler="template"> ❷
        <![CDATA[
        Welcome to NEFS, ${vc.fieldStates.getState("first_name").value.textValue} ❸
        ${vc.fieldStates.getState("last name").value.textValue}.
        <p>Form1 demonstrated how you can take two
        pieces of data and execute an arbritrary
        <a href="http://www.freemarker.org">FreeMarker</a> template embedded in
        the project.xml file.
        ]]>
    </on-execute>
 </dialog>
</project>
```

❶ Declaring a dialog using "`exec-sample`" template as the dialog type.
❷ The `handler` attribute of `on-execute` tag specifies the "`template`" handler for this dialog. Sparx provides various handlers for the dialogs (forms): *include*, *mail*, *command*, *panels* and *style-sheet*.
❸ Retrieving the value of dialog (form) fields, from their value contexts.

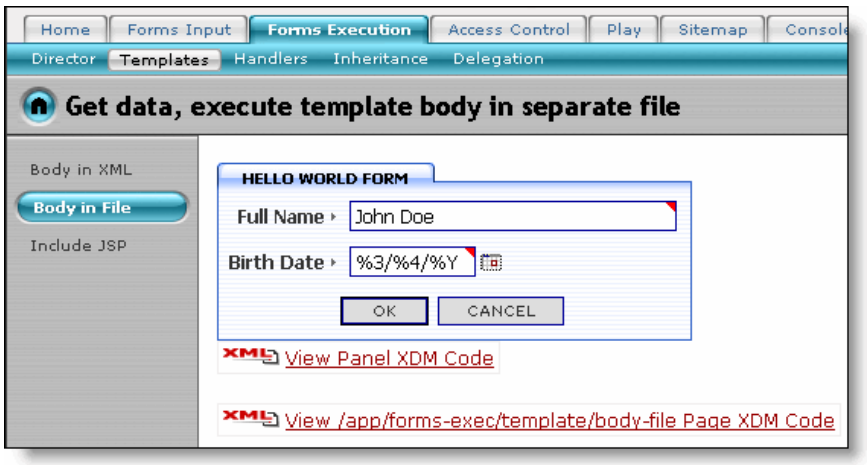
Note

In the above example, the template is embedded within `project.xml`

8.3.2.2. Body in File

This page contains an example of a dialog which will execute a template body defined in a seperate file and display the output.

Figure 19. Sample Page for Templates using Body in a Separate File



This example shows a Welcome message, obtained from a separate file, to the user. It also explains the loop attribute of dialog tag. To return to the dialog, you will need to click on the *Return to dialog* link provided on the page (as shown below):

Figure 20. Returning to the Dialog

The XML Code for using the FreeMarker™™ template (available in a separate file) is given below:

Example 4. XML Code for Using Template (in a separate file)

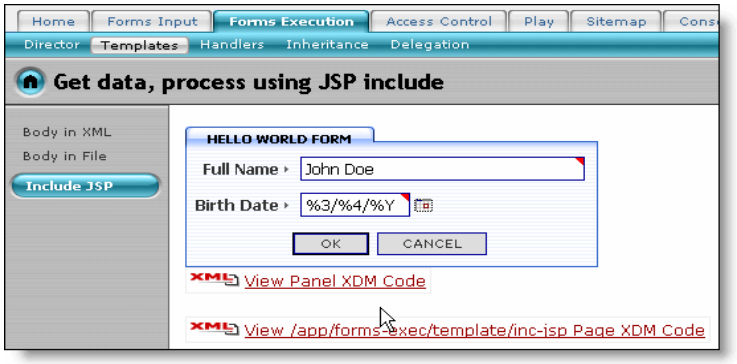
```
<project>
...
 <dialog type="exec-sample" name="body-file" loop="no"> ❶
    <frame heading="Hello World Form"/>
    <field type="text" name="full name" caption="Full Name" required="yes"/>
    <field type="date" name="birth date" caption="Birth Date" required="yes"
                       popup-calendar="yes"/>

    <on-execute handler="template">
        <source>form/exec/template-body-file.ftl</source> ❷
    </on-execute>
 </dialog>
...
</project>
```

❶ The `loop` attribute of `dialog` tag is used to specify not to show the dialog after the dialog executes.
❷ `Source` tag is used to specify the template source file to be used (`template-body-file.ftl` in this case)

8.3.2.3. Body in JSP

This page contains an example of a dialog which will display an output from a JSP upon execution.

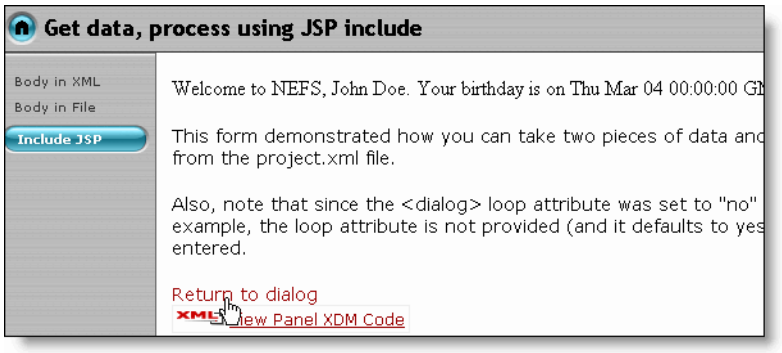
Figure 21. Sample Page for Templates using Body in a JSP File



This example shows a Welcome message, obtained from a JSP file, to the user. It also explains the loop attribute of dialog tag. To return to the dialog, you will need to click on the *Return to dialog* link provided on the page (as

shown below):

Figure 22. Returning to the Dialog



The XML Code for using the jsp file is given below:

Example 5. XML Code for Using a JSP File

```
<project>
...
 <dialog type="exec-sample" name="inc-jsp" loop="no">
    <frame heading="Hello World Form"/>
    <field type="text" name="full_name" caption="Full Name" required="yes"/>
    <field type="date" name="birth date" caption="Birth Date" required="yes"
           popup-calendar="yes"/>

    <on-execute handler="include"> ❶
        <path>/jsp/template-inc-test.jsp</path> ❷
    </on-execute>
 </dialog>
...
</project>
```

❶ The `handler` attribute of `on-execute` tag specifies the "`include`" handler for this dialog.
❷ `Path` tag is used to specify the JSP file to be used (*template-inc-test.jsp* in this case)

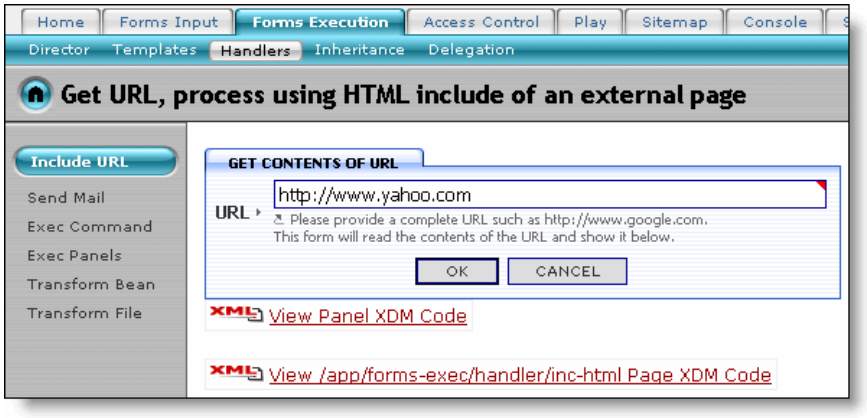
8.3.3. Handlers

This page demonstrates the usage of various *handlers* for the execution of a dialog. These handlers are processed once the internal validation of the dialog is complete and the dialog is ready to be be executed. Handlers are special built-in actions defined for convenience and reusability: *url*, *mail*, *command*, *panels*, and *style-sheet*. For defining more complex actions and logic for execution, please take a look at the *inheritence* or *delegation* examples.

8.3.3.1. Include URL

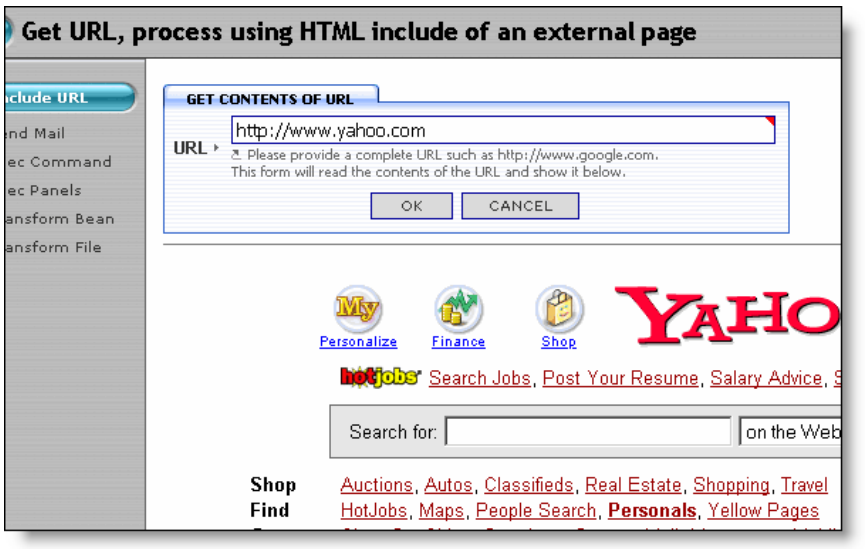
This page contains an example of a dialog which has a handler that will display HTML from an external page.

Figure 23. Sample Page for HTML Include of an External Page



Enter a URL and click the OK button. The contents of the target page, pointed to by this supplied URL, is displayed (as shown below):

Figure 24. Contents of the External Page



The XML Code for this example is given below:

Example 6. XML Code for Including an External Page

```
<project>
...
 <dialog type="exec-sample" name="inc-html" loop="prepend"
                            allow-multiple-executes="yes">
    <loop-separator>&lt;hr size=1&gt;&lt;p&gt;</loop-separator> ❶
    <frame heading="Get contents of URL"/>
    <field type="text" name="url" caption="URL" size="60" required="yes"
           default="http\://www.yahoo.com">
     <hint>
       Please provide a complete URL such as http\://www.google.com.&lt;br&gt;
       This form will read the contents of the URL and show it below.
     </hint>
    </field>

    <on-execute handler="include"> ❷
```

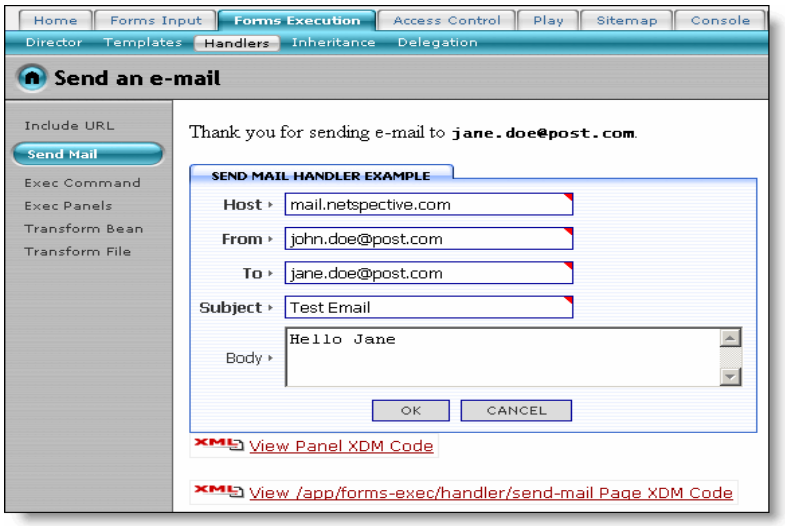
```
      <url>dialog-field:url</url> ❸
    </on-execute>
 </dialog>
...
</project>
```

❶ The loop-separator tag specifies a horizontal row to be displayed between the dialog and the external page content.
❷ The `handler` attribute of `on-execute` tag specifies the "include" handler for this dialog.
❸ `The URL` tag is used to specify the URL of the external page. The `dialog-field` static value specifies that the URL is obtained from the dialog field named *URL*

8.3.3.2. Send Mail

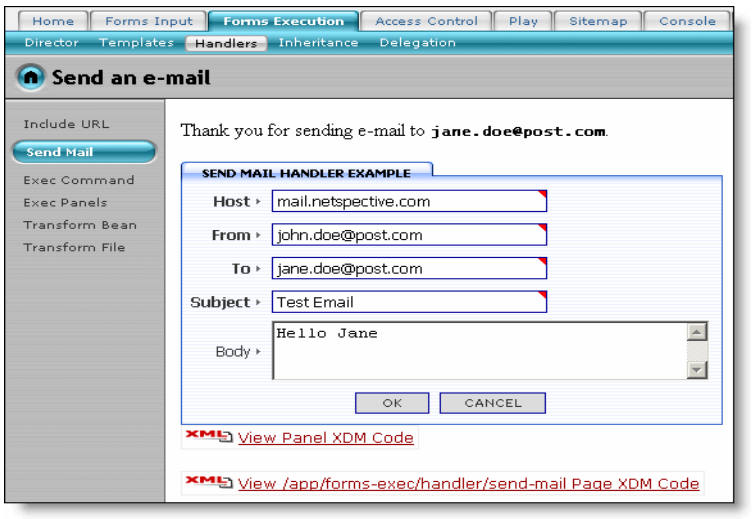
This page contains an example of a dialog which has a handler that will send an email upon execution of the dialog.

Figure 25. Sample Page for Send Mail



Enter the values for the mail dialog fields and click the **OK** button. A success message is displayed, along with the body of the email, if the mail is sent to the specified recipient. In case of an error, an appropriate error message is displayed.

Figure 26. Mail Send - Success Message



The XML Code for this example is given below:

Example 7. XML Code for Using the Mail Handler

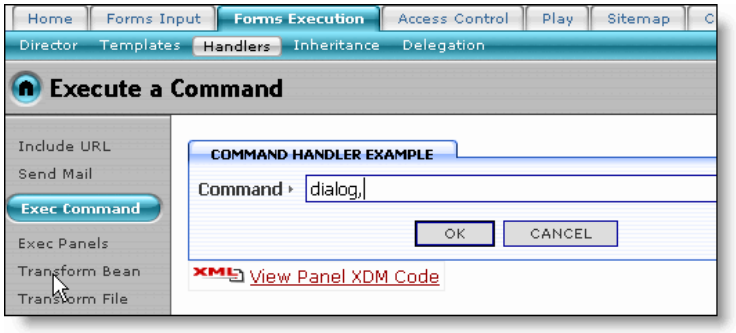
```
<project>
...
 <dialog type="exec-sample" name="send-mail" loop="append"
                                          allow-multiple-executes="yes">
    <frame heading="Send Mail Handler Example"/>
    <field type="text" name="host" caption="Host" required="yes"
                                  default="mail.netspective.com"/>
    <field type="text" name="from" caption="From" required="yes"
                                  default="dummy@mail.com"/>
    <field type="text" name="to" caption="To" required="yes"/>
    <field type="text" name="subject" caption="Subject" required="yes"/>
    <field type="memo" name="body" caption="Body"/>

    <on-execute handler="mail"> ❶
        <host>dialog-field:host</host> ❷
        <from>dialog-field:from</from>
        <to>dialog-field:to</to>
        <subject>dialog-field:subject</subject>
        <body>You entered this e-mail body:
                 ${vc.fieldStates.getState("body").value.textValue}</body> ❸
        <success-message> ❹
            <![CDATA[
            Thank you for sending e-mail to <code><b>
                ${vc.fieldStates.getState("to").value.textValue}</b></code>.
            ]]>
        </success-message>
        <failure-message> ❺
            <![CDATA[
            Unable to send e-mail to <code><b>
                ${vc.fieldStates.getState("to").value.textValue}</b></code>.<p>
            Exception:<br>
            <pre>${exception}</pre> ❻
            ]]>
        </failure-message>
    </on-execute>
 </dialog>
...
</project>
```

❶ The `handler` attribute of `on-execute` tag specifies the "mail" handler for this dialog.
❷ `The host` tag is used to specify the address for the mailing host. The `dialog-field` static value specifies that the mailing Host address is obtained from the dialog field named *host.*
❸ Displays the email's body text.
❹ The success-message tag specifies the message to be displayed when the email is successfully sent to the supplied recipient.
❺ The failure-message tag specifies the message to be displayed when the email is not sent to the supplied recipient due to some error.
❻ Displays the details of exception, in the error message.

8.3.3.3. Exec Command

This page contains an example of a dialog which has a handler that will process a command upon execution of the dialog. Enter the command and click the **OK** button to see the command execution result.

Figure 27. Execute Command



The XML Code for this example is given below:

Example 8. XML Code for Using the Command Handler

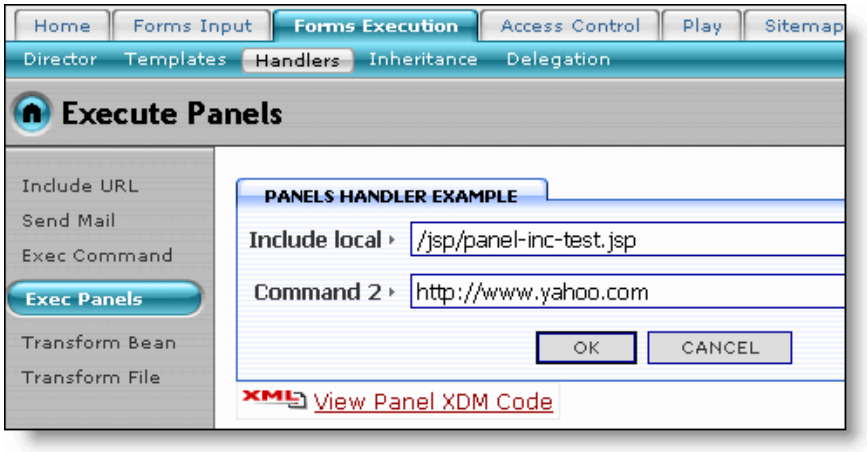
```
<project>
...
<dialog type="exec-sample" name="exec-cmd" allow-multiple-executes="yes">
    <frame heading="Command Handler Example"/>
    <field type="text" name="command" caption="Command" required="yes" size="60"/>

    <on-execute handler="command" command-expr="dialog-field:command"/> ❶
</dialog>
...
</project>
```

❶ The `handler` attribute of `on-execute` tag specifies the "command" handler for this dialog. The `command-expr` field specifies the dialog field named `command` as the source for the command expression.

8.3.3.4. Exec Panels

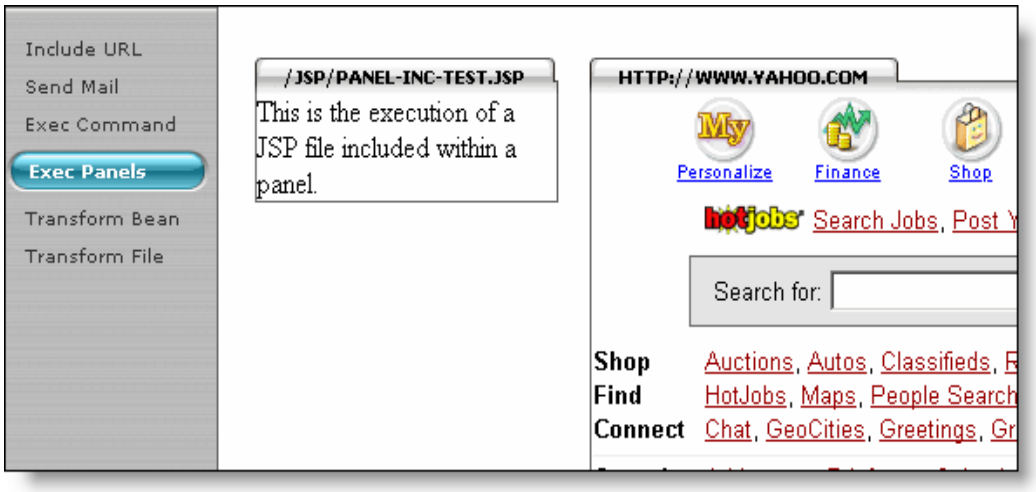
This page contains an example of a dialog which has a handler that will display panels upon execution of the dialog. The panels can contain HTML from various entities such as JSPs or URLs.

Figure 28. Sample Page for Execution of Panels



Provide a local source for HTML (`/jsp/panel-inc-test.jsp` in the above example) for `Include Local` field and a URL (`http://www.yahoo/com` in this case) as the value for `Command 2` field. Click the **OK** button. The HTML contents from the JSP and the yahoo web site are displayed in two different panels (as shown below):

Figure 29. HTML Loaded in Two Panels



The XML Code for this example is given below:

Example 9. XML Code for Using the Panels Handler

```
<project>
...
 <dialog type="exec-sample" name="exec-panels" allow-multiple-executes="yes">
    <frame heading="Panels Handler Example"/>
    <field type="text" name="local-include" caption="Include local"
           required="yes" size="60" default="/jsp/panel-inc-test.jsp"/>
    <field type="text" name="remote-include" caption="Command 2"
           required="yes" size="60" default="http\://www.yahoo.com"/>

    <on-execute handler="panels" style="two-columns"> ❶
        <panel type="include" path="dialog-field:local-include"> ❷
            <frame heading="dialog-field:local-include"/> ❸
        </panel>
        <panel type="include" URL="dialog-field:remote-include"> ❹
            <frame heading="dialog-field:remote-include"/>
        </panel>
    </on-execute>
 </dialog>
...
</project>
```

❶ The `handler` attribute of `on-execute` tag specifies the "panels" handler for this dialog. The `style` field specifies how the panels will be arranged in the output HTML. In this example, the panels will be arranged in two columns (one panel per column).

❷ Declaring the panel. The `type` attribute specifies the use of "`include`" template as the panel type. The `path` attribute uses a value-source to get the path from where the include file will be obtained. In the above example, this path value is obtained from the dialog field named `local-include`.

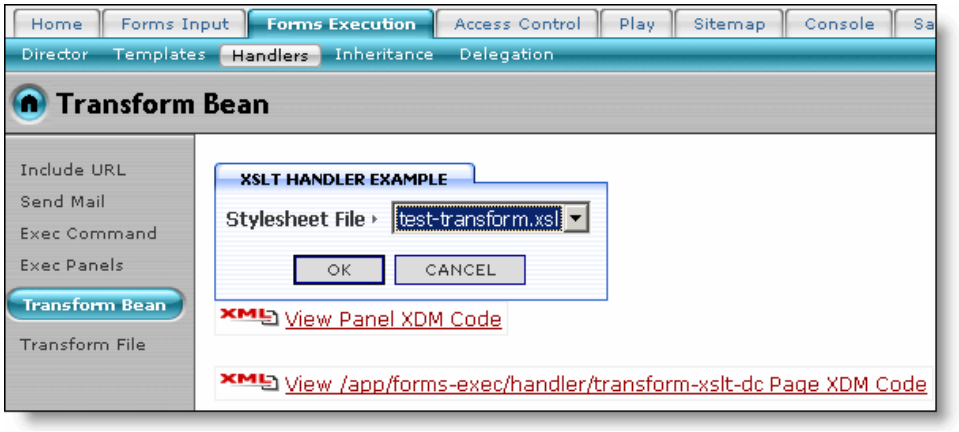
❸ The `heading` attribute of this frame specifies the value of the dialog-field named `local-include` as the heading for the frame.

❹ The `URL` attribute of this panel tag specifies the URL for the external page. In the above example, the value for the external page's URL is obtained from the dialog-field named `remote-include`.

8.3.3.5. Transform Bean

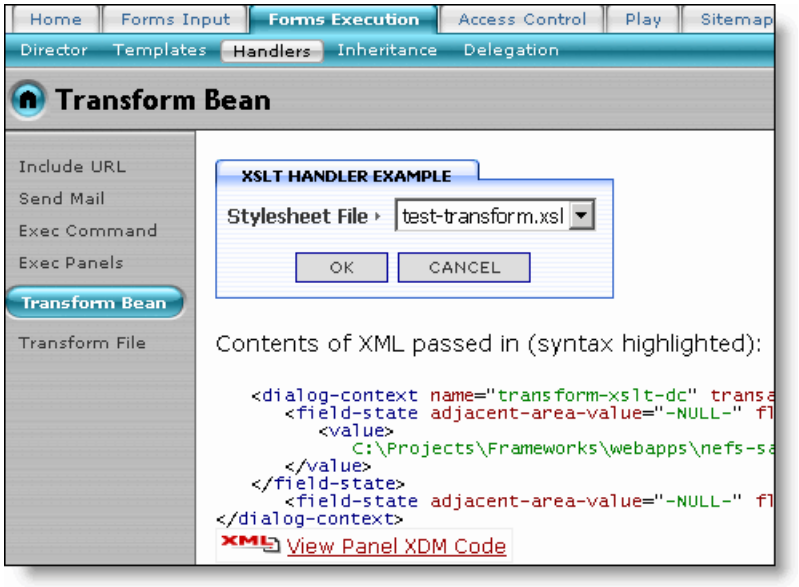
This page contains an example of a dialog which has a handler that will display output from a XSL style sheet transformation upon execution of the dialog.

Figure 30. Sample Page for XSL Transformation



The XSL style sheet is provided in the dialog. Click on the **OK** button to see the transformed XML describing the dialog's *Context Bean* (as shown below):

Figure 31. Transformed XML Describing the Dialog's Context Bean



The XML Code for this example is given below:

Example 10. XML Code for Using the Style-Sheet Handler (Transform Bean)

```
<project>
...
 <dialog type="exec-sample" name="transform-xslt-dc" loop="prepend"
                                                     allow-multiple-executes="yes">
  <frame heading="XSLT Handler Example"/>
   <field type="select" name="style-sheet-file" caption="Style sheet File"
          required="yes"
          choices="filesystem-entries:servlet-context-path:/form/exec,\.xsl$,yes"/> (1)

  <on-execute handler="style-sheet"> (2)
      <style-sheet-file>dialog-field:style-sheet-file</style-sheet-file> (3)
  </on-execute>
 </dialog>
...
</project>
```

(1) The `choices` attribute declares the list of XSL files from which the user can select. The `filesystem-entries` value-source specifies the list of all XSL files, available from the path specified by servlet context path, to be used as the entries for the select field.

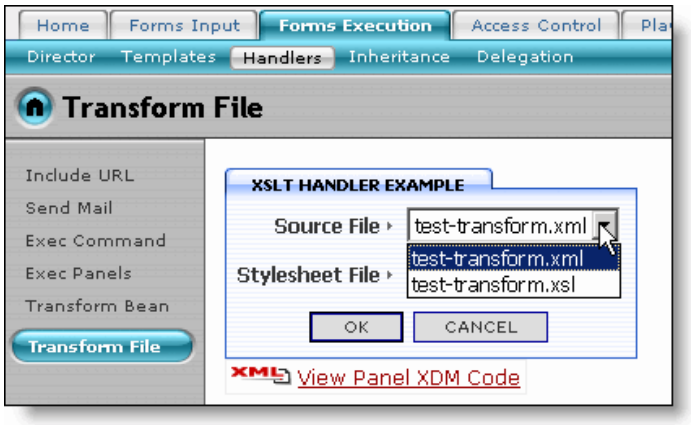
(2) The `handler` attribute of `on-execute` tag specifies the "style-sheet" handler for this dialog.

❸ The `style-sheet-file tag` specifies the XSL file to be used for transformation. In the above example, this value is obtained from the `style-sheet-file` field of the dialog.

8.3.3.6. Transform File

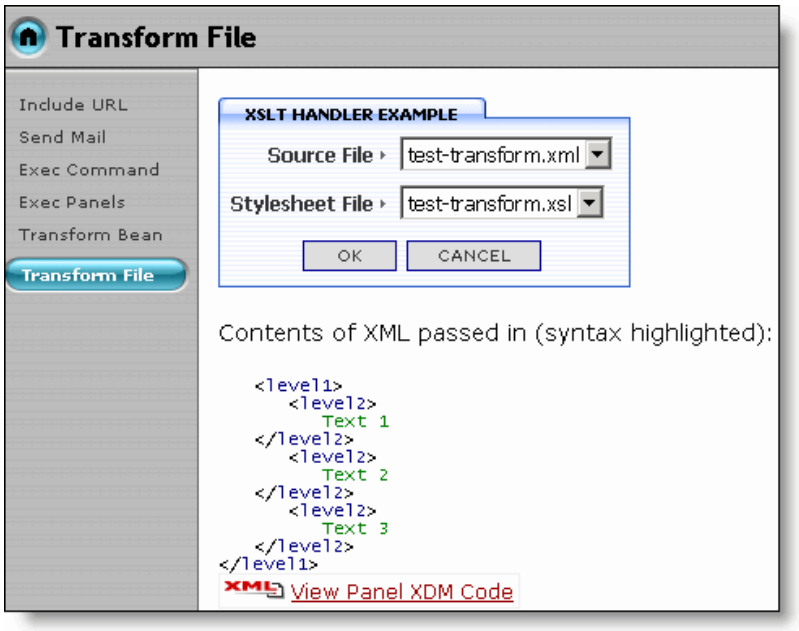
This page contains an example of a dialog which has a handler that will display output from a XSL style sheet transformation upon execution of the dialog.

Figure 32. Sample Page for XSL Transformation



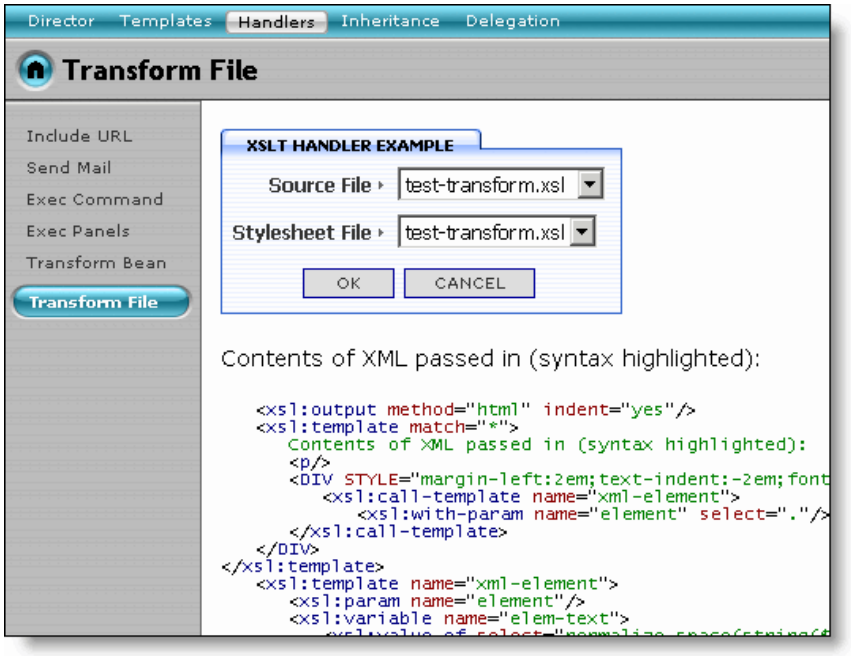
This dialog asks for a source XML or XSL file to be converted. The XSL to be applied can also be selected using the `Stylesheet File` select field in the dialog. Select the XML `test-transform.xml` file as the `Source File` and click the **OK** button. The transformed XML is displayed (as shown below):

Figure 33. Transformed XML File



You may also test the transformation of the XSL source file itself by selecting it as the source file. The transformed XSL file is displayed (as shown below):

Figure 34. Transformed XSL File



The XML Code for this example is given below:

Example 11. XML Code for Using the Style-Sheet Handler (Transform File)

```
<project>
...
 <dialog type="exec-sample" name="transform-xslt-file" loop="prepend"
                                                       allow-multiple-executes="yes">
    <frame heading="XSLT Handler Example"/>
    <field type="select" name="source-file" caption="Source File" required="yes"
           choices="filesystem-entries:servlet-context-path:/form/exec,\.x[sm]l$,yes"/>
    <field type="select" name="style-sheet-file" caption="Stylesheet File"
           required="yes"
           choices="filesystem-entries:servlet-context-path:/form/exec,\.xsl$,yes"/>

    <on-execute handler="style-sheet">
        <source-file>dialog-field:source-file</source-file> ❶
        <style-sheet-file>dialog-field:style-sheet-file</style-sheet-file>
    </on-execute>
 </dialog>
...
</project>
```

❶ The `source-file tag` specifies the file on which XSL transformation is applied.

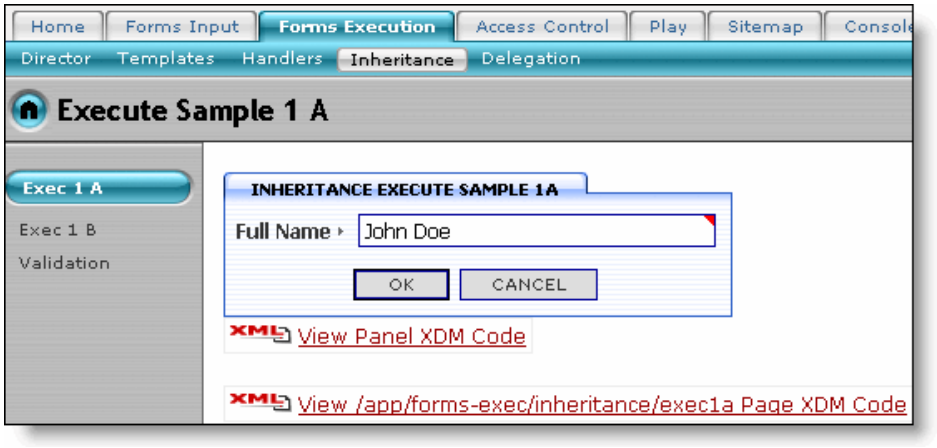
8.3.4. Inheritance

This page demonstrates the *inheritance* model for defining custom logic or action for different stages of a dialog. This model should only be used for the most complex cases where you need to override the default behavior of the dialog. By extending the default dialog class, com.netspective.sparx.form.Dialog, you have full control over the dialog and its behavior.

8.3.4.1. Exec 1 A

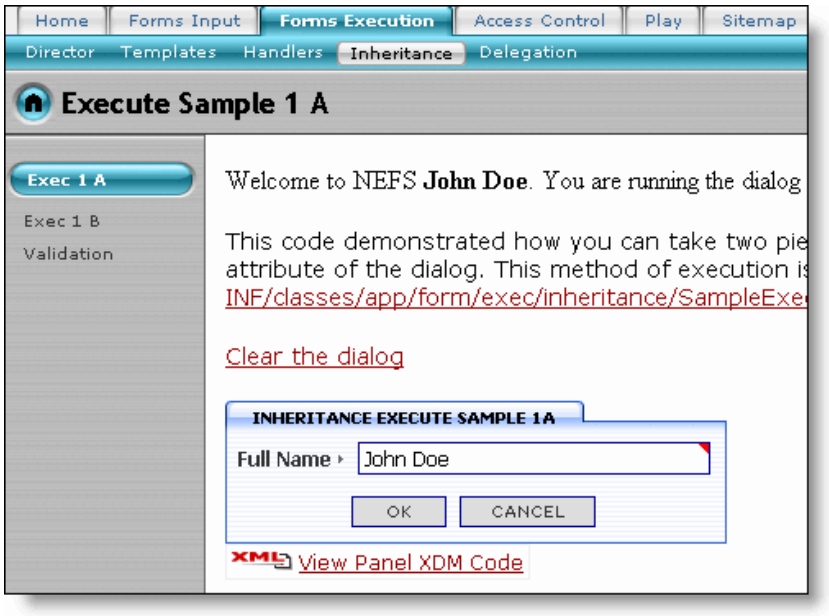
This page contains a dialog which has a custom dialog class that overrides only the default execution behavior of the dialog.

Figure 35. Sample Page for Handling Dialog 1A (through Inheritance)



Enter a value for `Full Name` field and click the **OK** button. The custom Java class (`SampleExecuteDialog` in this case) displays a *Welcome* message along with the supplied name (as shown below):

Figure 36. Welcome Message Generated by the Custom Class



The welcome message also displays the link to the custom class being used as the dialog handler.

The XML Code for this example is given below:

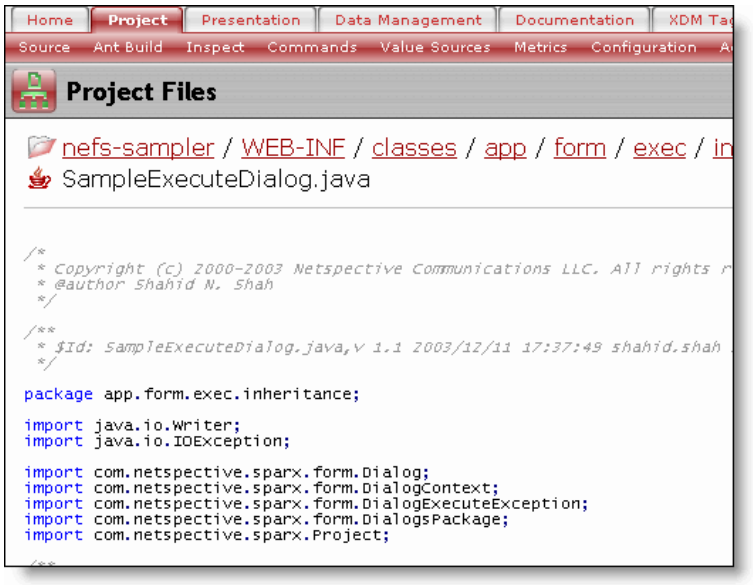
Example 12. XML Code for Using Custom Handler (Inheritance)

```
<project>
...
 <dialog type="exec-sample" name="exec1a"
                           class="app.form.exec.inheritance.SampleExecuteDialog"> ❶
    <frame heading="Inheritance Execute Sample 1a"/>
    <field type="text" name="full_name" caption="Full Name" required="yes"/>
 </dialog>
...
</project>
```

❶ the `class` attribute of the dialog tag specifies the custom class (`SamplerExecuteDialog` in this case) to be used as the handler for this dialog.

You may view the code for custom Java class using the link provided in the welcome message. This opens the Java source code in the Console (as shown below):

Figure 37. Source File for Custom Class (Inheritance)



Example 13. Custom Dialog Handler Class (using Inheritance)

```
package app.form.exec.inheritance; (1)

       (2)
import java.io.Writer;
import java.io.IOException;

import com.netspective.sparx.form.Dialog;
import com.netspective.sparx.form.DialogContext;
import com.netspective.sparx.form.DialogExecuteException;
import com.netspective.sparx.form.DialogsPackage;
import com.netspective.sparx.Project;

public class SampleExecuteDialog extends Dialog (3)
{
    public SampleExecuteDialog(Project project)
    {
        super(project);
    }

    public SampleExecuteDialog(Project project, DialogsPackage pkg)
    {
        super(project, pkg);
    }

    public void execute(Writer writer, DialogContext dc)
                    throws IOException, DialogExecuteException (4)
    {
      writer.write("Welcome to NEFS <b>" +
            dc.getFieldStates().getState("full_name").getValue().getTextValue() + (5)
            "</b>. ");


      writer.write("You are running the dialog called <b>"+ dc.getDialog().getName()
            +"</b> in package <b> "+
            dc.getDialog().getNameSpace().getNameSpaceId() +"</b>.");

      if(dc.getDialog().getQualifiedName().equals("form.exec.inheritance.exec1b")) (6)
          writer.write("<p>You are <b>"+
                        dc.getFieldStates().getState("age").getValue().getIntValue()
                        + "</b> years old.");

      writer.write("<p>");


      writer.write("This code demonstrated how you can take two pieces of data and " +
                   "execute an arbritrary class by setting the 'class' attribute of "+
                   "the dialog. ");
```

```
      writer.write("This method of execution is called <i>Dialog Inheritance</i>. ");

      String relativePath =
             /WEB-INF/classes/app/form/exec/inheritance/SampleExecuteDialog.java"; ❼


      String sourceFileLink =
dc.getConsoleFileBrowserLinkShowAlt(dc.getServlet().getServletConfig().getServletContext
().getRealPath(relativePath),relativePath); ❽


      writer.write("You may review the code at " + sourceFileLink + ".");
      writer.write("<p>");
      writer.write("<a href=\""+ dc.getNavigationContext().getActivePage().getName() +
                   "\">Clear the dialog</a>"); ❾
    }
}
```

❶ Declaring the package for this class.
❷ Importing other required classes.
❸ Extending the `Dialog` class to provide custom implementation.
❹ Declaring the dialog `execute` method which is called as soon as all dialog data is entered and validated.
❺ Retrieving the value of the dialog field named full_name from the dialog context and sending it to the writer.
❻ Checking the dialog name after retrieving it from the dialog context. If the dialog name is exec1b, the Age related message is also displayed in the output HTLML.
❼ Declaring a variable containing the relative path of this Java file.
❽ Getting the link to the source file of custom dialog handler class (`SampleExecuteDialog.java` in this case). The `getConsoleFileBrowserLinkShowAlt` method shows the name of the file that is passed into the first parameter. If the second parameter is also passed, the URL (anchor) reference shows an alternate text within the anchor.
❽ Retrieving the current page's name from the Navigation Context.

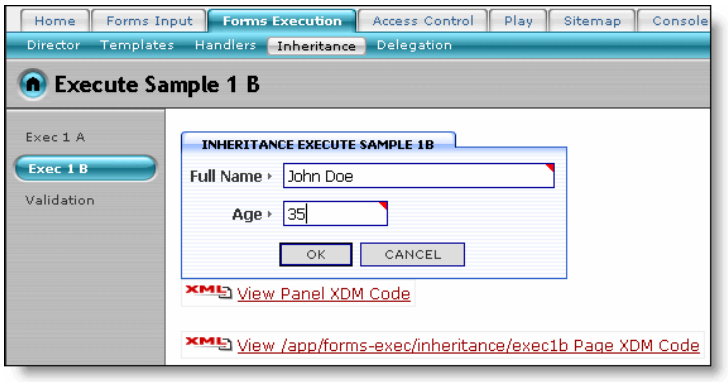
Note

This custom dialog handler class takes care of `exec1A` and `exec2a` both, in case of Sampler App.

8.3.4.2. Exec 1 B

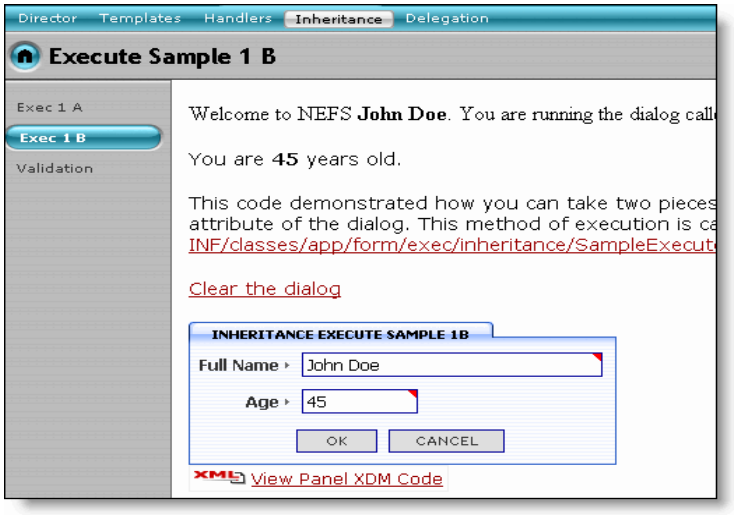
This page contains a dialog which has a custom dialog class that overrides only the default execution behavior of the dialog.

Figure 38. Sample Page for Handling Dialog 1B (through Inheritance)



Enter a value for `Full Name` and `Age` fields and click the **OK** button. In the Sampler App, the custom Java class `SampleExecuteDialog` handles this dialog as well. Note the message about the supplied age (as shown below):

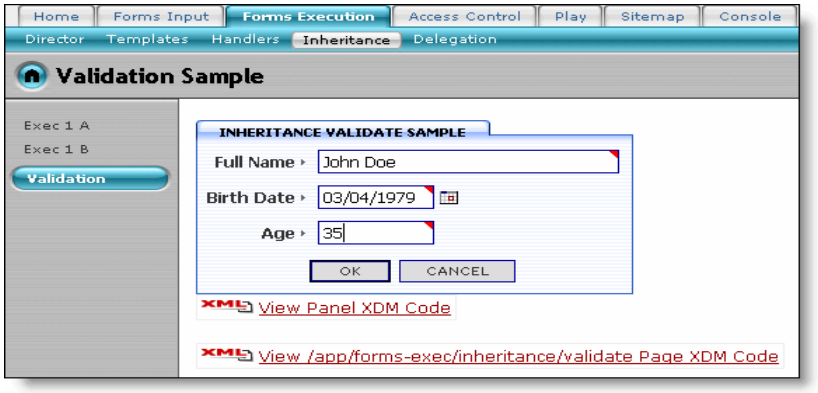
Figure 39. Welcome Message Generated by the Custom Class



8.3.4.3. Validation

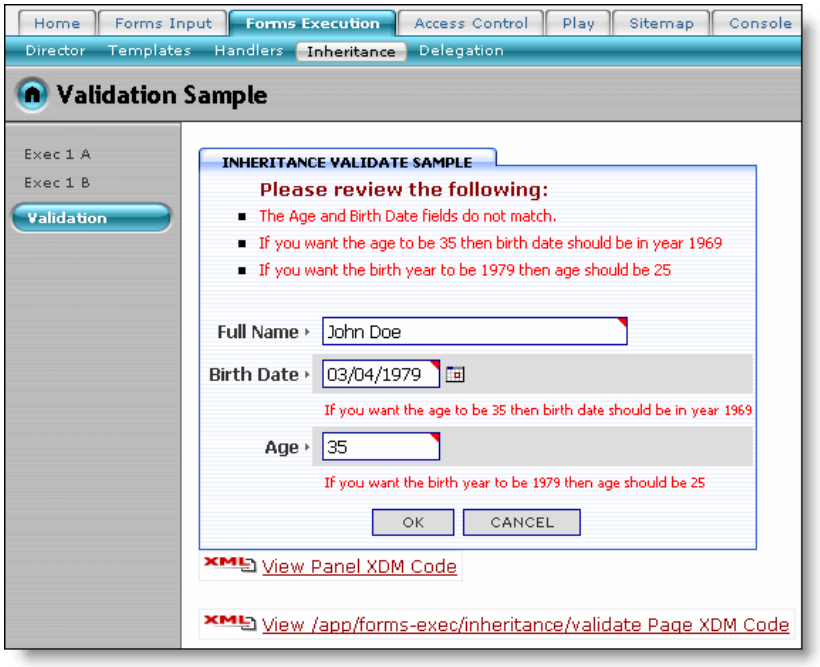
This page contains a dialog which has a custom dialog class that overrides the default population, validation, and execution behaviors of the dialog.

Figure 40. Sample Page for Validation using Custom Class (through Inheritance)



Enter the values for the dialog fields and click the **OK** button (as shown in the image above). The custom validation class (`SampleValidateDialog` in this case) checks the entered values and displays the validation messages, if any (as shown below):

Figure 41. Validation Messages Generated by Custom Class



The XML Code for this example is given below:

Example 14. XML Code for Validation Custom Class (Inheritance)

```
<project>
...
 <dialog type="exec-sample" name="validate"
         class="app.form.exec.inheritance.SampleValidateDialog"
         generate-dcb="yes"> ❶
    <frame heading="Inheritance Validate Sample"/>
    <field type="text" name="full_name" caption="Full Name" required="yes"/>
    <field type="date" name="birth_date" caption="Birth Date" required="yes"
                                                              max="today"/>
    <field type="integer" name="age" caption="Age" required="yes" min="1" max="150"/>
 </dialog>
...
</project>
```

❶ Declaring the custom class handling the dialog (form) validation. The `generate-dcb` attribute specifies the generation of auto-generated Dialog Context Bean (DCB).

The code for custom validation class is given below. Note that this class extends the `Dialog` class to provide custom implementation.

Example 15. Custom Dialog Validation Handler Class (using Inheritance)

```
package app.form.exec.inheritance;

import java.io.Writer;
import java.io.IOException;
```

```
import java.util.Date;
import java.util.Calendar;
import auto.dcb.form.exec.inheritance.ValidateContext;

import com.netspective.sparx.form.Dialog;
import com.netspective.sparx.form.DialogContext;
import com.netspective.sparx.form.DialogExecuteException;
import com.netspective.sparx.form.DialogsPackage;
import com.netspective.sparx.form.field.DialogField;
import com.netspective.sparx.Project;

public class SampleValidateDialog extends Dialog
{
  public SampleValidateDialog(Project project)
 {
     super(project);
 }

 public SampleValidateDialog(Project project, DialogsPackage pkg)
 {
     super(project, pkg);
 }

 public void populateValues(DialogContext dc, int formatType) ❶
 {
     super.populateValues(dc, formatType);

     if(dc.getDialogState().isInitialEntry() &&
        formatType == DialogField.DISPLAY FORMAT) ❷
     {
         ValidateContext validateContext = new ValidateContext(dc); ❸
         Date birthDate = validateContext.getBirthDate().getDateValue();
         if(birthDate == null)
         {
             Calendar cal = Calendar.getInstance();
             // assume we're 25 years old
             cal.set(Calendar.YEAR, cal.get(Calendar.YEAR) - 25);

             validateContext.getBirthDate().setValue(cal.getTime());
         }
     }
 }

 public boolean isValid(DialogContext dc) ❹
 {
     if(! super.isValid(dc)) ❺
         return false;
      ValidateContext validateContext = new ValidateContext(dc);
      Date birthDate = validateContext.getBirthDate().getDateValue();
      int age = validateContext.getAge().getIntValue();

                    ❻
      Calendar cal = Calendar.getInstance();
      int currentYear = cal.get(Calendar.YEAR);
      cal.setTime(birthDate);
      int yearOfBirth = cal.get(Calendar.YEAR);

      if(currentYear - yearOfBirth != age)
      {
       dc.getValidationContext().addError("The Age and Birth Date fields "+
                                          "do not match."); ❼

                                  ❽
       validateContext.getBirthDateField().invalidate(dc, "If you want the age to be "+
                age + " then birth date should be in year " + (currentYear - age));
       validateContext.getAgeField().invalidate(dc, "If you want the birth year to be
```

```
       "+ yearOfBirth +" then age should be " + (currentYear - yearOfBirth));

       return false; ❾
      }

      return true;
 }

 public void execute(Writer writer, DialogContext dc)
                                        throws IOException, DialogExecuteException ❿
 {
  writer.write("Congratulations <b>"
               + dc.getFieldStates().getState("full name").getValue().getTextValue() +
               "</b>! ");


  writer.write("It seems that your birthdate and ages match correctly.");
  writer.write("<p>");
  writer.write("This code demonstrated how you can take two three pieces of data, "+
              "have Sparx individually validate the general fields such as text, "+
              "date, and integer but add your own custom validator some other logic"+
              " by overriding the isValid() method of the "+
              "com.netspective.sparx.form.Dialog class.<p>");

  String relativePath = "/WEB-INF/classes/app/inheritance/SampleValidateDialog.java";


  writer.write("You may review the code at " +
dc.getConsoleFileBrowserLinkShowAlt(dc.getServlet().getServletConfig().getServletContext
().getRealPath(relativePath), relativePath));


  writer.write("<p>");
  writer.write("<a href=\""+
  dc.getNavigationContext().getActivePage().getName() +"\">Clear the dialog</a>");
 }
}
```

❶ This method is called each time the dialog is displayed to the user and fields need their values populated. By default, Sparx handles populates almost all the value population. But, sometimes, you may want to populate something programmatically.

❷ Only put data into the fields if we're displaying the data for the first time. If it's not the initial entry, it means that there's an error on the screen and the fields need to be populated with the data entered by the user.

❸ Using the auto-generated dialog context bean (DCB) for user data retrieval. DCB is a type-safe wrapper around the general-purpose `DialogContext`. Declaring a validation context for current user's dialog.

❹ The `isValid` method is called right before `execute` to validate the dialog. If you need to perform any validation that the fields can't perform on their own (such as cross-field validation), you should perform it here. The `dc` parameter contains reference to the `DialogContext` containing the field values and states.

❺ Check if any of the default validation failed. If yes, leave this method.

❻ Find out the current year and the year of birth entered by the user.

❼ Add an error message, at the dialog level, if the custom validation condition fails. The error message is added to the validation context of the dialog context.

❽ In case of failure, add error messages to each field as well.

❾ Return "false" so the dialog does not go into execute mode until you have valid data.

❿ The dialog `execute` method is called as soon as all data is entered and validated. This method is guaranteed to be called *only* when all fields' data is valid.

Note

This custom dialog validation handler class (using inheritance) takes care of `exec1A` and `exec2a` both, in case of Sampler App.

8.3.5. Delegation

This page demonstrates the delegation model for defining custom logic or action for different stages of a dialog. The Framework allows definition of various listeners for different stages of a dialog. By implementing the listener interfaces and then assigning the custom listeners to a dialog, these listeners will get executed.

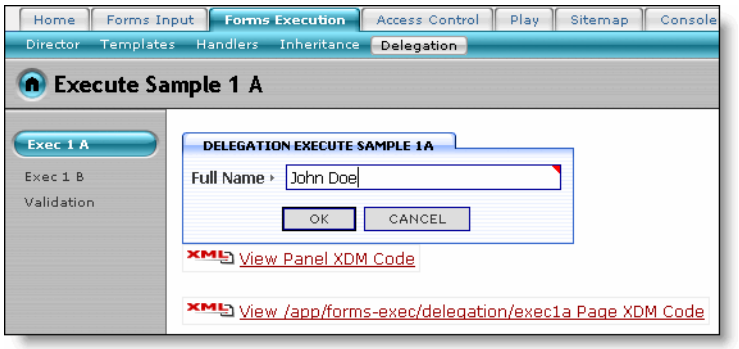
Note

Inheritance is used for large-scale customizations and enhancing the framework itself while delegation is used to implement listeners and event processing customizations. Although both methods are supported, the delegation model is recommened to help ensure that changes in the NEF APIs do not affect your code.

8.3.5.1. Exec 1 A

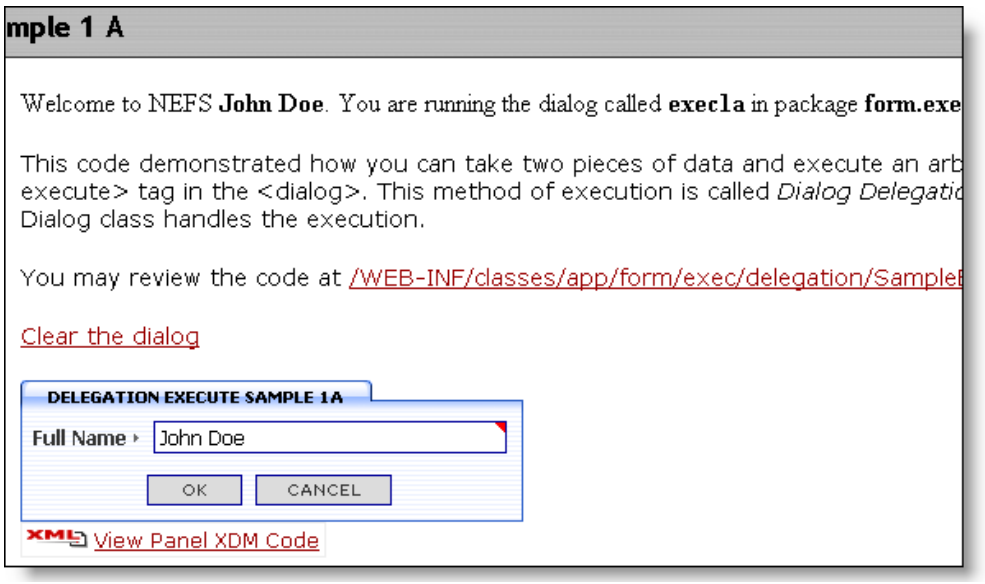
This page demonstrates an *execution* listener for the *delegation* model. An execution listener implements the com.netspective.sparx.form.handler.DialogExecuteHandler interface.

Figure 42. Sample Page for Handling Dialog 1A (through Delegation)



Enter a value for `Full Name` field and click the **OK** button. The custom Java class (`SampleExecuteHandler` in this case) displays a *Welcome* message along with the supplied name (as shown below):

Figure 43. Welcome Message Generated by the Custom Class



The XML Code for this example is given below:

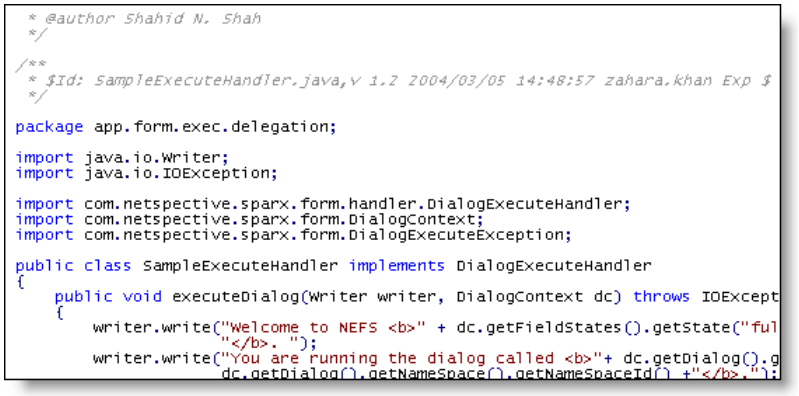
Example 16. XML Code for Using Custom Handler (Delegation)

```
<project>
...
 <dialog type="exec-sample" name="exec1a">
    <frame heading="Delegation Execute Sample 1a"/>
    <field type="text" name="full name" caption="Full Name" required="yes"/>

    <on-execute class="app.form.exec.delegation.SampleExecuteHandler"/> ❶
 </dialog>
...
</project>
```

❶ the `class` attribute of the `on-execute` tag specifies the custom class (`SampleExecuteHandler` in this case) to be used as the listener for this dialog.

You may view the code for custom Java class using the link provided in the welcome message. This opens the Java source code in the Console (as shown below):

Figure 44. Source File for Custom Class (Delegation)



```
 * @author Shahid N. Shah
 */

/**
 * $Id: SampleExecuteHandler.java,v 1.2 2004/03/05 14:48:57 zahara.khan Exp $
 */

package app.form.exec.delegation;

import java.io.Writer;
import java.io.IOException;

import com.netspective.sparx.form.handler.DialogExecuteHandler;
import com.netspective.sparx.form.DialogContext;
import com.netspective.sparx.form.DialogExecuteException;

public class SampleExecuteHandler implements DialogExecuteHandler
{
    public void executeDialog(Writer writer, DialogContext dc) throws IOExcept
    {
        writer.write("Welcome to NEFS <b>" + dc.getFieldStates().getState("ful
                     "</b>. ");
        writer.write("You are running the dialog called <b>"+ dc.getDialog().g
                     dc.getDialog().getNameSpace().getNameSpaceId() +"</b>.");
```

Example 17. Custom Dialog Handler Class (using Delegation)

```
package app.form.exec.delegation; ❶

    ❷
import java.io.Writer;
import java.io.IOException;

import com.netspective.sparx.form.handler.DialogExecuteHandler;
import com.netspective.sparx.form.DialogContext;
import com.netspective.sparx.form.DialogExecuteException;

public class SampleExecuteHandler implements DialogExecuteHandler ❸
{
    public void executeDialog(Writer writer, DialogContext dc)
                              throws IOException, DialogExecuteException ❹
    {
     writer.write("Welcome to NEFS <b>" +
     dc.getFieldStates().getState("full name").getValue().getTextValue()+ "</b>. "); ❺

      writer.write("You are running the dialog called <b>"+ dc.getDialog().getName() +
       "</b> in package <b>"+ dc.getDialog().getNameSpace().getNameSpaceId() +
       "</b>.");

      if (dc.getDialog().getQualifiedName().equals("form.exec.delegation.exec1b")) ❻
```

```
        writer.write("<p>You are <b>"+
                     dc.getFieldStates().getState("age").getValue().getIntValue()
                     + "</b> " + "years old.");

        writer.write("<p>");
        writer.write("This code demonstrated how you can take two pieces of data and "+
                     "execute an arbritrary class by supplying an &lt;on-execute&gt; "+
                     "tag in the &lt;dialog&gt;. ");
        writer.write("This method of execution is called <i>Dialog Delegation</i> "+
                     "because a class separate from the Dialog class handles the "+
                     "execution.<p>");

        String relativePath =
             "/WEB-INF/classes/app/form/exec/delegation/SampleExecuteHandler.java"; ❼


       String sourceFileLink =
dc.getConsoleFileBrowserLinkShowAlt(dc.getServlet().getServletConfig().getServletContext
().getRealPath(relativePath), relativePath); ❽


        writer.write("You may review the code at " + sourceFileLink + ".");
        writer.write("<p>");
        writer.write("<a href=\""+ dc.getNavigationContext().getActivePage().getName()
                       +"\">Clear the dialog</a>"); ❾
    }
}
```

❶ Declaring the package for this class.
❷ Importing other required classes.
❸ Implementing the `DialogExecuteHandler` interface.
❹ Declaring the dialog `executeDialog` method which is called as soon as all dialog data is entered and validated.
❺ Retrieving the value of the dialog field named full_name from the dialog context and sending it to the writer.
❻ Checking the dialog name after retrieving it from the dialog context. If the dialog name is exec1b, the Age related message is also displayed in the output HTLML.
❼ Declaring a variable containing the relative path of this Java file.
❽ Getting the link to the source file of custom dialog handler class (`SampleExecuteHandler.java` in this case). The `getConsoleFileBrowserLinkShowAlt` method shows the name of the file that is passed into the first parameter. If the second parameter is also passed, the URL (anchor) reference shows an alternate text within the anchor.
❾ Retrieving the current page's name from the Navigation Context.

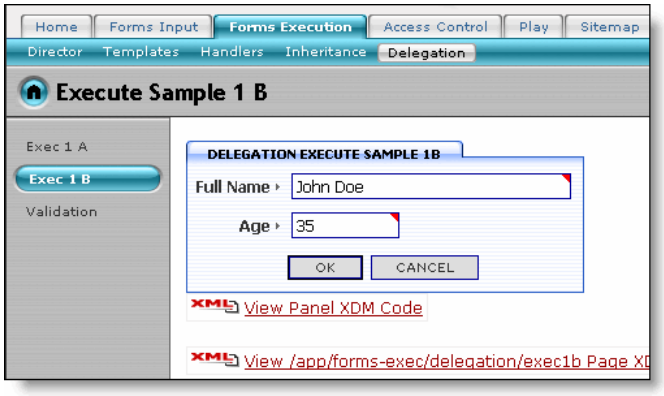
Note

This custom class takes care of `exec1A` and `exec2a` both, in case of Sampler App.

8.3.5.2. Exec 1 B

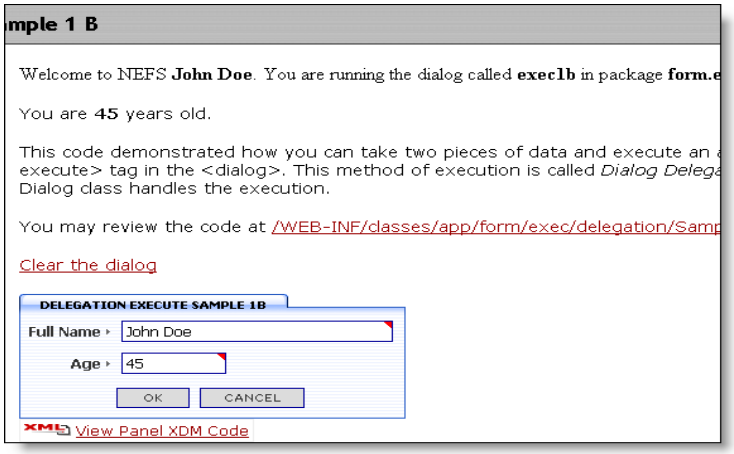
This page demonstrates an *execution* listener for the *delegation* model. An execution listener implements the com.netspective.sparx.form.handler.DialogExecuteHandler interface.

Figure 45. Sample Page for Handling Dialog 1B (through Delegation)



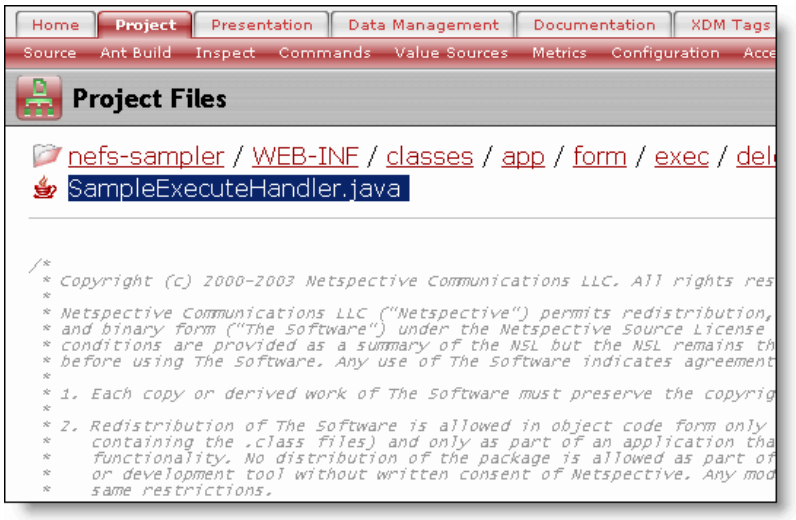
Enter a value for `Full Name` and `Age` fields and click the **OK** button. In the Sampler App, the custom Java class `SampleExecuteHandler` handles this dialog as well. Note the message about the supplied age (as shown below):

Figure 46. Welcome Message Generated by the Custom Class



You may view the code for custom Java class using the link provided in the message. This opens the Java source code in the Console.

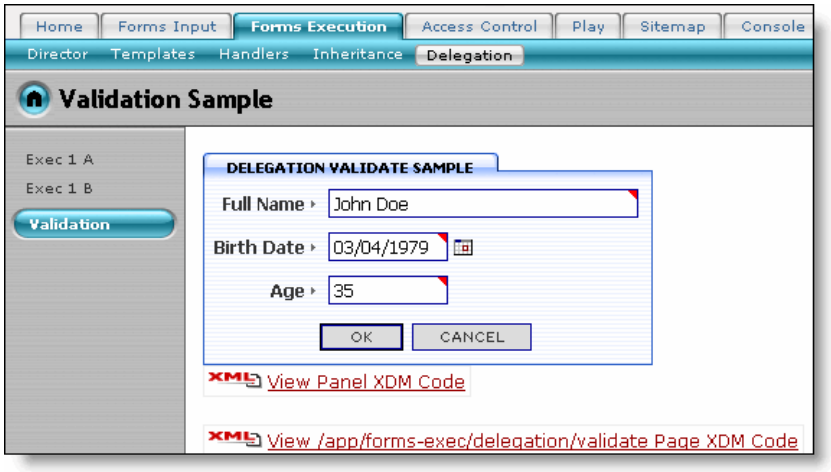
Figure 47. Source File for Custom Class



8.3.5.3. Validation

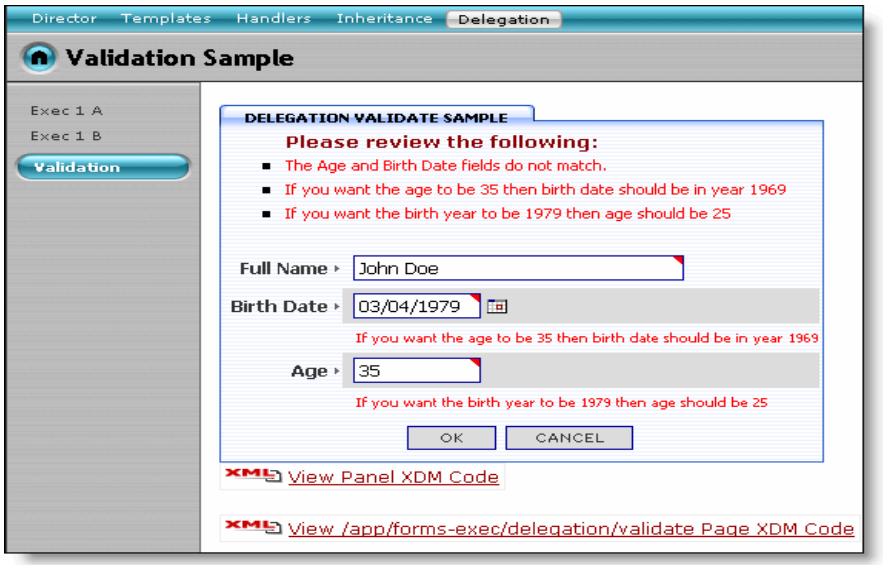
This page contains a dialog that demonstrates *validation*, *population*, and *execution* listeners for the *delegation* model. A listener class can implement various listener interfaces to define custom business logic for various dialog stages.

Figure 48. Sample Page for Validation using Custom Class (through Delegation)



Enter the values for the dialog fields and click the **OK** button (as shown in the image above). The custom validation class (`SampleValidateDialog` in this case) checks the entered values and displays the validation messages, if any (as shown below):

Figure 49. Validation Messages Generated by Custom Class



The XML Code for this example is given below:

Example 18. XML Code for Validation Custom Class (Delegation)

```
<project>
...
 <dialog type="exec-sample" name="validate" generate-dcb="yes"> ❶
    <frame heading="Delegation Validate Sample"/>
    <field type="text" name="full name" caption="Full Name" required="yes"/>
    <field type="date" name="birth_date" caption="Birth Date" required="yes"
                                                              max="today"/>
```

```
    <field type="integer" name="age" caption="Age" required="yes" min="1" max="150"/>

    <listener class="app.form.exec.delegation.SampleValidateHandler"/> ❷
 </dialog>

...
</project>
```

❶ The `generate-dcb` attribute specifies the generation of auto-generated Dialog Context Bean (DCB).
❷ The `class` attribute of the `listener` tag declares the dialog listener to be used for custom validation.

The code for custom validation class, using delegation, is given below. Note that this class implements the `DialogValidateListener`, `DialogPopulateListener` and `DialogExecuteHandler` interfaces to provide custom implementation.

Example 19. Custom Dialog Validation Handler Class (using Delegation)

```
package app.form.exec.delegation;

import java.io.Writer;
import java.io.IOException;
import java.util.Date;
import java.util.Calendar;

import auto.dcb.form.exec.delegation.ValidateContext;

import com.netspective.sparx.form.DialogContext;
import com.netspective.sparx.form.DialogExecuteException;
import com.netspective.sparx.form.DialogValidationContext;
import com.netspective.sparx.form.handler.DialogExecuteHandler;
import com.netspective.sparx.form.listener.DialogValidateListener;
import com.netspective.sparx.form.listener.DialogPopulateListener;
import com.netspective.sparx.form.field.DialogField;

public class SampleValidateHandler implements DialogValidateListener,
                                               DialogPopulateListener,
                                               DialogExecuteHandler
{
 public void populateDialogValues(DialogContext dc, int formatType) ❶
 {
  if(dc.getDialogState().isInitialEntry() && formatType==DialogField.DISPLAY_FORMAT) ❷
  {
    ValidateContext validateContext = new ValidateContext(dc); ❸
    Date birthDate = validateContext.getBirthDate().getDateValue();
    if(birthDate == null)
    {
     Calendar cal = Calendar.getInstance();
     // assume we're 25 years old
     cal.set(Calendar.YEAR, cal.get(Calendar.YEAR) - 25);
     validateContext.getBirthDate().setValue(cal.getTime());
    }
  }
 }


 public void validateDialog(DialogValidationContext dvc) ❹
 {
   ValidateContext validateContext = new ValidateContext(dvc.getDialogContext());
   Date birthDate = validateContext.getBirthDate().getDateValue();
   int age = validateContext.getAge().getIntValue();



            ❺
   Calendar cal = Calendar.getInstance();
   int currentYear = cal.get(Calendar.YEAR);
   cal.setTime(birthDate);
   int yearOfBirth = cal.get(Calendar.YEAR);

   if(currentYear - yearOfBirth != age)
   {
    dvc.addError("The Age and Birth Date fields do not match."); ❻
```

```
    validateContext.getBirthDateField().invalidate(dvc.getDialogContext(), "If you "+
                "want the age to be "+ age +" then birth date should be in year " +
                 (currentYear - age)); ❼


    validateContext.getAgeField().invalidate(dvc.getDialogContext(), "If you want "+
                " the birth year to be " + yearOfBirth +" then age should be " +
                 (currentYear - yearOfBirth));
  }
}
public void executeDialog(Writer writer, DialogContext dc)
            throws IOException, DialogExecuteException ❽
{
  writer.write("Congratulations <b>" +
               dc.getFieldStates().getState("full name").getValue().getTextValue()
               + "</b>! ");
  writer.write("It seems that your birthdate and ages match correctly.");
  writer.write("<p>");
  writer.write("This code demonstrated how you can take two three pieces of data, "+
               "have Sparx individually validate the general fields such as text, "+
               "date, and integer but add your own custom validator some other "+
               "logic by creating listeners for various events such as population,"+
               " validation, and execution.<p>");

  String relativePath =
            "/WEB-INF/classes/app/form/exec/delegation/SampleValidateHandler.java";


  writer.write("You may review the code at " + dc.getConsoleFileBrowserLinkShowAlt(
   dc.getServlet().getServletConfig().getServletContext().getRealPath(relativePath),
                                                          relativePath));


  writer.write("<p>");
  writer.write("<a href=\""+ dc.getNavigationContext().getActivePage().getName()
               +"\">Clear the dialog</a>");
}

}
```

❶ Declaring custom implementation for `populateDialogValues` method of `DialogPopulateListener`. This method is called each time the dialog is displayed to the user and fields need their values populated. By default, Sparx handles populates almost all the value population. But, sometimes, you may want to populate something programmatically.

❷ Only put data into the fields if we're displaying the data for the first time. If it's not the initial entry, it means that there's an error on the screen and the fields need to be populated with the data entered by the user.

❸ Using the auto-generated dialog context bean (DCB) for user data retrieval. DCB is a type-safe wrapper around the general-purpose `DialogContext`. Declaring a validation context for current user's dialog.

❹ The `validateDialog` method is called right before `execute` to validate the dialog. If you need to perform any validation that the fields can't perform on their own (such as cross-field validation), you should perform it here. The `dvc` parameter contains reference to the `DialogValidationContext` containing the dialog's validation context.

❺ Find out the current year and the year of birth entered by the user.

❻ Add an error message, at the dialog level, if the custom validation condition fails. The error message is added to the dialog validation context.

❼ In case of failure, add error messages to each field as well.

❽ The `executeDialog` method is called as soon as all data is entered and validated. This method is guaranteed to be called *only* when all fields' data is valid.

Note

This custom dialog validation handler class (using delegation) takes care of `exec1A` and `exec2a` both, in case of Sampler App.

8.4. Access Control

This module is for demonstrating how the Access Control feature can be used. This feature is for setting up roles and capabilities to handle controlling access to various parts of an application. An application user can be

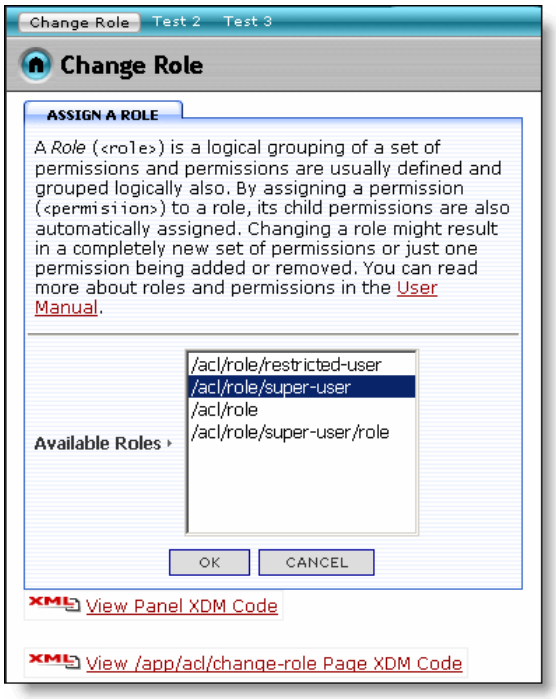
assigned a role which will control whether or not the user can access a page, a report, a dialog, or even a field.

The examples contained in this section are further divided into sub-categories. The *Access Control* page contains a list of links to its children pages containing the categorized example pages. Links to these children pages are also provided through a sub menu bar under the *Access Control* tab.

8.4.1. Change Role

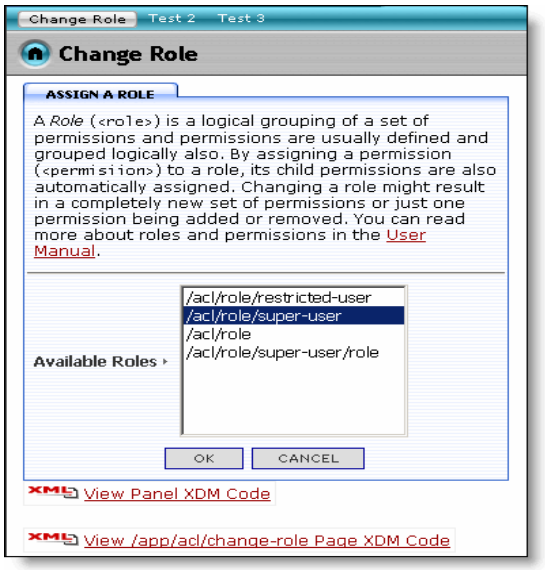
This page is for changing a Role assigned to the user. A Role (`role`) is a logical grouping of a set of permissions and permissions are usually defined and grouped logically also. By assigning a permission (`permission`) to a role, its child permissions are also automatically assigned. Changing a role might result in a completely new set of permissions or just one permission being added or removed.

Figure 50. Changing User Roles



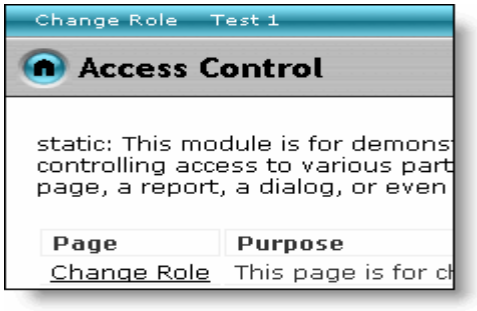
Select a new user role from the `Available Roles` select box and click the **OK** button to assign a new role to the user.

Figure 51. Role Changed



The links visible in the Navigation Menu bar change based on the selected Role. When you change the role from `acl/role/super-user` to `acl/role/super-user/role`, the navigation menu bar links change from Test2 and Test3 to Test1.

Figure 52. Navigation Menu Changed After Role Change

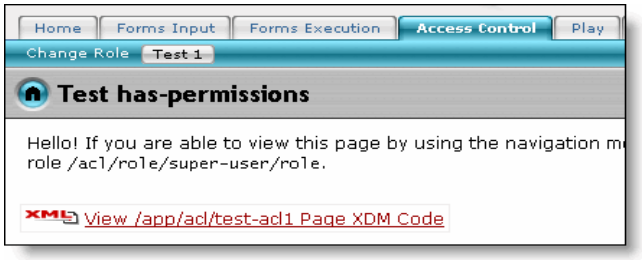


Test the Access Control option by further changing the role and observing the change in the links visible on the Navigation Menu bar.

8.4.2. Test 1

This page is hidden in the navigation menu when the user has the following permission: /acl/app/forms/execution. Clicking on this link from the Navigation Menu bar displays a message informing you about the permission rights currently available:

Figure 53. Permission Available Message



When this link is not visible in the navigation menu tab, you may select it from the *Access Control* page.

8.4.3. Test 2

This page is hidden in the navigation menu when the user lacks the following permission: /acl/app/forms/execution. You may click on this link from the Access Control page. You may also test it from the Navigation Menu bar when it is visible there.

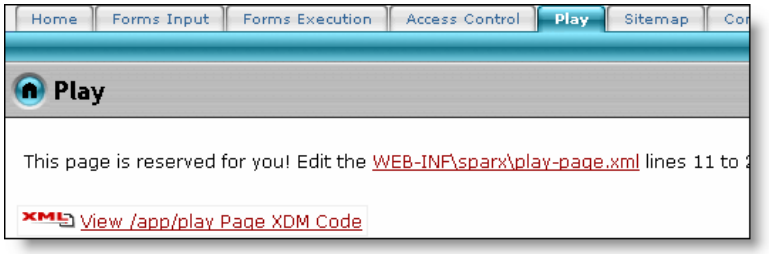
8.4.4. Test 3

This page is hidden in the navigation menu when the user lacks the following permission: /acl/app/forms. You may click on this link from the Access Control page. You may also test it from the Navigation Menu bar when it is visible there.

8.5. Play

Now that you have learnt the use of some more features of the framework, it is time for you to start creating your own applications! The Play page gives you a chance to learn by writing your own code. You can do this by updating the WEB-INF\sparx\play-page.xml file. The Play page provides you the opportunity to test the functionality of your newly written code.

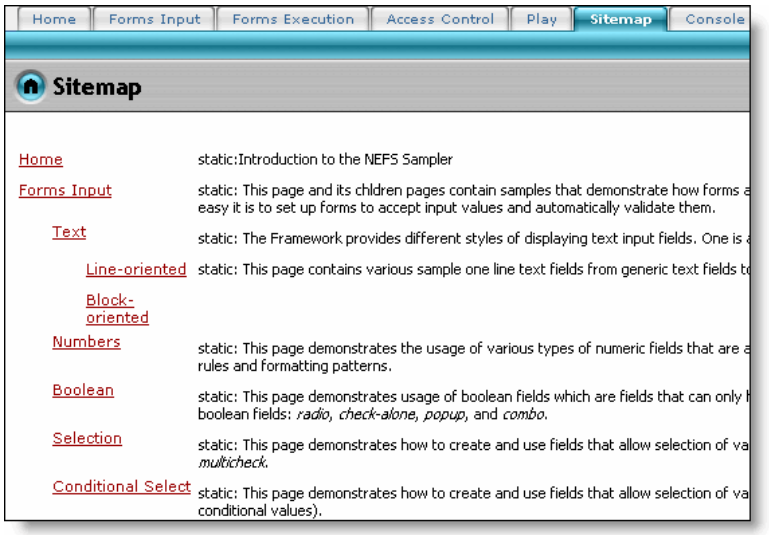
Figure 54. Page for Creating and Testing Your Own Code



8.6. Sitemap

Sparx provides you with the facility to create sitemaps for your web sites using templates. The *Sitemap* page demonstrates this facility by providing a sitemap for the Sampler App.

Figure 55. Sampler App Sitemap



8.7. Console

Like every other NEFS application, the Sampler also has a Console. This Console can be accessed through the Console section. You will see the application Console login screen.

The Console's default User Id is `'console'` and the default Password is `'console'`(each without quotes). Unless otherwise specified, that is the User Id and Password combination you should use if the Console prompts you to login. Please refer to the *Enterprise Console Tour* for more details about the Console.

8.8. Sample Apps Home

This section takes you to the Sample Apps Home page on the Netspective web site.

9. Conclusion

Congratulations! You have explored some of the core concepts provided by NEFS. You have learnt how to create your dialogs and input fields using Sparx. You have seen examples describing a variety of powerful Sparx components to build your web applications. You can now build your own applications that incorporate custom validation, custom dialog handlers and access control. To learn about more advance concepts, try the other sample applications.