Ericsson Catalog Manager and Ericsson Order Care

Realize Higher Consistency for Faster Time-to-Revenue

Tips for Effective Use

12/1553-CSH 109 420/1C Uen Rev A

Contents

1 Introduction

This document provides tips on using the Ericsson Order Care software effectively.

1.1 Purpose and Scope

This guide provides tips on using the Ericsson Order care software to change how content is displayed in applications that you are designing. These tips include creating and displaying dynamic content, embedding metadata actions and other design considerations for setting how content is displayed in an application.

1.2 Reader’s Guideline

This section describes the version syntax covered in this document and any additional, required information.

Commands that you enter on the command line appear in courier font, such as the following:

```
svnadmin dump C:\SVN\myProject > C:\backupFolder\myProject.bak
```

Document names and sections within documentation are set in italics, such as the following:

For more information on making a copy of your project metadata, see the *Velocity Studio User Guide*, under *Velocity Studio User Interface > Common Actions Outside Velocity Studio*.

Note: To navigate the documentation, an arrow appears (>), which separates each hyperlink to be clicked.

2 Modifying Application Content

2.1 Overview

Setting the content to display in an application can be achieved in different ways. This document shows examples of the different approaches to this task.

2.2 Modification by application initialization

1 Create new metadata and a new namespace.

2 Under this namespace, create a new metadata object of your choice. For this example, a top level user interface will be created.

3 To create a top level user interface, right click on the namespace tree node. Select **New** > **User Interface**. Call this user interface “byInitScriptUI ”.

4 Override the Default form for byInitScriptUI. Add a simple label or header to indicate that this user interface is being set as the application’s content through the init script.

5 Create an application user interface. Right click on the namespace tree node. Select **New** > **User Interface**. Call this user interface “modificationByInitScript”. Be sure to set Overrides as com.conceptwave.system.Application.

6 Once your application is created, go to the **Methods** tab. You will see a brown book icon . Click this icon. You are then presented with inherited methods. Right-click the onInit() method and select **Override**.

   a After the line “this.cw$super_onInit();”, create a new instantiation of byInitScriptUI and return it. The script below will return a new instance and set it as the content variable value of the application.

   ```
   (return new namespaceName.userInterfaceName(model,
   parent))

   return new contentModification.byInitScriptUI(null,this);
   ```

   b Run Framework. Once you’ve selected this application, byInitScriptUI should be displayed automatically.

2.3 Modification by menu script

2.3.1 1.3.1 By return script

1. Under the same namespace, create a new metadata object of your choice. For this example, a top level user interface will be created.

2. To create a top level user interface, right click on the namespace tree node. Select **New** > **User Interface**. Call this user interface "byMenuScriptUI ".

3. Override the Default form for byMenuScriptUI. Add a simple label or header to indicate that this user interface is being set as the application's content through a menu's script.

4. Create an application user interface. Right click on the namespace tree node. Select **New** > **User Interface**. Call this user interface "modificationByMenuScript". Be sure to set Overrides as com.conceptwave.system.Application.

5. Once your application is created, navigate to the **Methods** tab. Right click on the modificationByMenuScript tree node and select New User Action Method.

   a. In the script portion of the method, create a new instantiation of byMenuScriptUI and return it. The script below will return a new instance and set it as the content variable value of the application:

   ```
   return new contentModification.byMenuScriptUI(null,this);
   ```

6. Override the Menu form for modificationByMenuScript.

7. Add a new menu to HLayoutForMenuItems.

   a. Set a label, and set its "Click method" property to the method created in step 5.
   b. Run Framework. Once you've selected this application, click on the menu you created in step 6. byMenuScriptUI should be displayed automatically.

2.3.2 By variable value

1. Repeat steps 1 to 5 of part A. In step 5, change the method so that instead of returning a new object, you are setting the content variable's value to the new object:

   ```
   this.content = new
   contentModification.byMenuScriptUI(null,this);
   ```

   ```
   this.contentForm = "Default"; //it is important to set this
   line, otherwise, no content will be shown
   ```

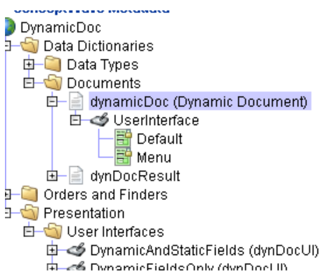
2. Continue to Steps 6 and 7 of part A.

3 Creating and displaying a Dynamic Document

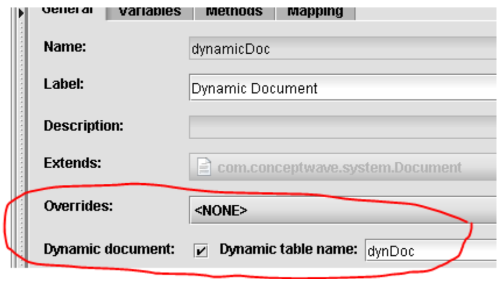
1. Create a table in your database to store the dynamic data in. This table must have the following columns in order. The first four columns are the unique key of the table:

   a DOC_ID VARCHAR2(36) -- contains the GUID or ID of the document
   b DOCUMENT_TYPE VARCHAR2(38) -- GUID of the document type
   c LEAF_NAME VARCHAR2(64) -- the leaf name
   d ARRAY_INDEX NUMBER(4) -- for single value leaves is 0, for arrays is from 1 to n
   e DATA_TYPE NUMBER(2) -- 0 – String, 1 – Number, 2 – Boolean
   f VALUE NVARCHAR2(256) -- The value stored as string as defined above

2. In Velocity Studio, create a document with dynamic flag set to true. Provide the name of the table you want to use for the dynamic data.



DynamicDoc
- Data Dictionaries
  - Data Types
  - Documents
    - dynamicDoc (Dynamic Document)
      - UserInterface
        - Default
        - Menu
    - dynDocResult
- Orders and Finders
- Presentation
  - User Interfaces
    - DynamicAndStaticFields (dynDocUI)



Name: dynamicDoc
Label: Dynamic Document
Description:
Extends: com.conceptwave.system.Document
Overrides: <NONE>
Dynamic document: ✓ Dynamic table name: dynDoc

3. To display the dynamic document, create a top level user interface.

4. In the Variables tab, add a variable of type dynamicDoc (the document created in step 2).

5. Navigate to the Methods tab and override the onInit() function.

6. Initialize the document variable using the following syntax:

```
new
DynamicDocument(docFullName,docId,optionalThrowException,run
LoadScript)
```

   Example:

```
var docVar
newDynamicDocument("DynamicDoc:dynamicDoc","5",false,false);
```

7. Now create some document leaf using the following syntax:

```
createLeaf(leafName, leafLabel, type, arraySize, length,
precision,codeTable);
```

Example:

```
docVar.createLeaf("testDecLeafName", "testDecLeaf", 0,
0,4,0, null);
docVar.createLeaf("testIntLeafName", "testIntLeaf", 1,
0,4,0, null);
docVar.createLeaf("testStringLeafName", "testStringLeaf", 3,
0,4,0, null);
docVar.createLeaf("testBooleanLeafName", "testBooleanLeaf",
4, 0,4,0, null);
docVar.createLeaf("testDateLeafName", "testDateLeaf", 6,
0,4,0, null);
```

Note: If this testcase is to be viewed multiple times, be sure that either the database is emptied OR use the following code to determine that the leaf name is not already being used:

```
if( docVar.isDynamicLeaf("testDecLeafName")==null)
docVar.createLeaf("testDecLeafName", "testDecLeaf", 0,
0,4,0, null);
```

8 Override the default form of the user interface created in Step 3.

9 Add a Vertical stack layout element. Right click on the layout and click **Add**.

10 Select **Structures** > **Dynamic Document** and click **Finish**.

11 Set the dynamic document element's "Variable" property to the variable you added in Step 4.

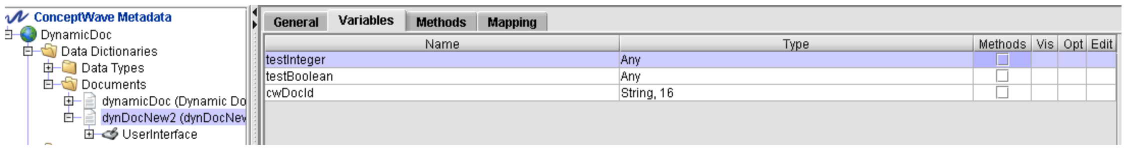
12 Add a menu element in your application menu to display this user interface.

13 Save and run. When the menu is clicked in runtime, you should see one field for each leaf created in Step 7.

3.1 Dynamic Document with predefined layout

1 Create a new document with the Dynamic Doc flag set to true. Provide the database table to use at runtime.

2 Navigate to the Variables tab of the document and add variables of type “Any” (or if type is already determined, set the proper type) with the **same name as ones that can be found in the database table**.



Name	Type	Methods	Vis	Opt	Edit
testInteger	Any	☐			
testBoolean	Any	☐			
cwDocId	String, 16	☐			

Create a new top level user interface and add a variable to the dynamic document.

3 Override the `onInit()` method and initialize your variable.

Example:

```
this.dynamicDocNew = new
DynamicDocument("DynamicDoc.dynDocNew2",
test5",false,false);
```

4 Navigate to the Default form and override. Add a VerticalStackLayout.

5 Under this layout, add a Dynamic Document element.

6 Right click on the dynamic document element and add the desired input elements.

7 You may choose to display other elements under the dynamic document such as images or buttons that are not associated with the dynamic document.

8 Design the forms as desired.

9 Save and run.

3.2 Saving a user entry with Dynamic Document

1 Continuing from the application create in the above example, navigate to the user interface created in Step 3.

2 In the Methods tab, create a new user action called "save". In this script, only one line is needed:

```
this.docVar.save();
```

- Navigate to the default form and add a new menu element. Set its click method to the save method from Step 2.
- Remember to set the label or icon of this menu to clearly see it in runtime.
- Save and run. In runtime, edit the values of the fields and then click save. The new data should now be in the database table created in Step 1.

3.3 Displaying static variables with Dynamic Document

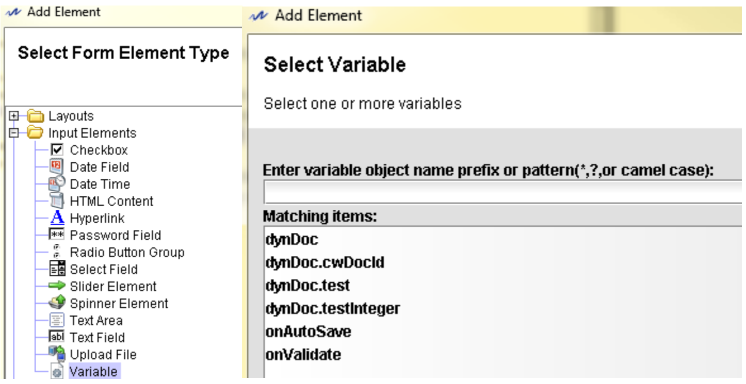
1 Continuing from the application create in the above example, navigate to the document created in Step 2.

2 In the **Variables** tab, add the necessary static variables. Be sure to map the document. The static part of this document needs a separate table in the database.

3 Navigate to the user interface default form created in Step 3.

4 Right click on the vertical stack and add the static fields (**Input Elements** > **Variables** or pick the form element type first then specify the variable to use with it).

5 Save and run. In runtime, both dynamic variables and static variables should be present. When saved, all input should be present in the database in the corresponding tables.



dynDoc
dynDoc.cwDocId
dynDoc.test
dynDoc.testInteger
onAutoSave
onValidate

3.4 Supported data types

Available data types that are supported include:

DECIMAL = 0;
INTEGER = 1;
STRING = 3;
BOOLEAN = 4;
DATE TIME = 5;
DATE = 6;
CODE TABLE = 9

4 Navigation bar validation

4.1 Overview

This section demonstrates different approaches to validating objects using the navigation bar. By default, the navigation bar does not validate the current object displayed.

For the purpose of this example, three documents will be used. Create these three documents and perform the necessary mapping. For each document, override the default form and create a layout that will display the variables. For noDataDoc, a simple label will suffice.

1. validateDoc
   variables: firstName; type String; Optional: false
   lastName; type String; Optional: false
   dateOfBirth: type Date;

2. noValidateDoc
   variables: firstName; type String; Optional: true
   lastName; type String; Optional: true
   dateOfBirth: type Date; Optional: true

3. noDataDoc

4.2 Automatic validation

This example will validate only when going to the next step. If the user is trying to go back a previous step, validation will not be called.

1 Create a new navigation bar called “automaticNavBar”.

2 In the **Variables** tab, create three variables of type **DocumentUserInterface**:
   a step1doc
   b step2doc
   c step3doc

3 In the **Methods** tab, create three methods:

   a onStep1:

```
 if(this.step1doc == null)

 this.step1doc = new
navigationBarValidation.noValidateDoc.UserInterface(null,
this);

 return this.step1doc;
```

   b onStep2:

```
 if(this.step2doc == null)

 this.step2doc = new
 navigationBarValidation.validateDoc.UserInterface(null,

this);return this.step2doc;
```

   c onStep3:

```
if(this.step3doc == null)

this.step3doc = new
navigationBarValidation.noValidateDoc.UserInterface(null,
this); return this.step3doc;
```

4 In the **Configuration** tab, create three navigation bar items: step1,step2,step3. At the minimum, specify the Click Method and the Default Image URL.

5 To gain the validation functionality for this example, go back to the Methods tab and override the setCurrentObject method.

6 Add the following line after the super implementation call. This flag with a null value indicates that the navigation bar should be validated going forward, and ignore validation when going back a step.

```
this.cwDoValidate  = null;
```

7 Create a new Application User Interface. In the Variables tab, add a variable of type automaticNavBar. Add a leafInitAction for this variable.

```
return new navigationBarValidation.automaticNavBar(null,
this);
```

8 Right click on the Page form and override. Set the page title.

9 Right click on the menu form frame (mLayout/menu) and click “Copy+Replace”. In the variable property, set to the navigation bar. In the Form property, set to automaticNavBar.Forms.HLayout

10 Right click on the content form frame and click “Copy+Paste”.
In the variable property, set as automaticNavBar.currentObject.
In the Form property, set to automaticNavBar.currentObject.Default (or the form to display).

11 Save and run. Click Step 1 to display step1doc. Notice that to go to Step 2, there are no validations as these fields are not mandatory. Going from Step2 to Step3 without filling out any information should not allow you to continue and will produce validation errors. If you try to go back to Step 1 with the validation errors still present, the UI should allow you to do so.

4.3 Per Object Validation

This example will validate the current object depending on the step the user is currently at.

1 Using the metadata created above, create a copy of validateDoc and name it “partialValidateDoc”.

2 Set the Optional property of the firstName to TRUE.

3 Create a new navigation bar called perObjectNavBar and create navigation bar items step1,step2 and step3. You may choose to copy and paste the navigation bar from above (be sure to delete “setCurrentObject” in the copy).

4 In the Methods tab, create three methods:

   a onStep1:

   ```
   if(this.step1doc == null)

   this.step1doc = new
   navigationBarValidation.noValidateDoc.UserInterface(null,
   this);

   this.setCurrentObject(this.step1doc, true);
   ```

   b onStep1:

   ```
   if(this.step2doc == null)

   this.step2doc = new
   navigationBarValidation.validateDoc.UserInterface(null,
   this);

   this.setCurrentObject(this.step2doc, false);
   ```

   c onStep3:

   ```
   if(this.step3doc == null)

   this.step3doc = new
   navigationBarValidation.noValidateDoc.UserInterface(null,
   this);

   this.setCurrentObject(this.step3doc, false);
   ```

5 setCurrentObject accepts two parameters, one is the object to display and the second parameter, whether to validate this object when the user navigates away from this step.

6 Create a new Application User Interface. In the **Variables** tab, add a variable of type perObjectNavBar. Add a leafInitAction for this variable.

   ```
   return new navigationBarValidation.perObjectNavBar(null,
   this);
   ```

7 Right click on the Page form and override. Set the page title.

8 Right click on the menu form frame (mLayout/menu) and click “Copy+Replace”. In the variable property, set to the navigation bar. In the Form property, set to perObjectNavBar.Forms.HLayout

9 Right click on the content form frame and click “Copy+Paste”. In the variable property, set as perObjectNavBar.currentObject. In the Form property, set to perObjectNavBar.currentObject.Default (or the form to display).

10 Save and run. Click Step 1 to display step1 doc. Notice that to go to Step 2, there are now validations. Going from Step1 to Step3 without filling out any information should not allow you to continue and will produce validation errors as well.

4.4 Validate and continue

This example will validate the current object but allow the user to continue to the next/previous step.

1 Create a new navigation bar called validateContinueNavBar and create navigation bar items step1,step2 and step3. You may choose to copy and paste the navigation bar from above.

2 In the Methods tab, create four methods:

a validateCurrentObject:

```
 if(this.getCurrentObject() != null){
 this.getCurrentObject().onValidate();
 }
```

b onStep1:

```
 if(this.step1doc == null)
    this.step1doc = new

navigationBarValidation.noValidateDoc.UserInterface(null,
this);

this.validateCurrentObject();
   this.setCurrentObject(this.step1doc, true);
```

c onStep2:

```
 if(this.step2doc == null)
    this.step2doc = new

navigationBarValidation.validateDoc.UserInterface(null,
this);

    this.validateCurrentObject();
   this.setCurrentObject(this.step2doc, true);
```

d onStep3:

```
 if(this.step3doc == null)
       this.step3doc = new

    navigationBarValidation.noValidateDoc.UserInterface(null
    , this);

    this.validateCurrentObject();
   this.setCurrentObject(this.step3doc, false);
```

3 `validateCurrentObject` validates the current object being displayed and shows any validation errors in the browser once you go back to that step.

4 Create a new Application User Interface. In the **Variables** tab, add a variable of type perObjectNavBar. Add a leafInitAction for this variable.

```
return new navigationBarValidation.perObjectNavBar(null,
this);
```

5 Right click on the Page form and override. Set the page title.

6 Right click on the menu form frame (mLayout/menu) and click “Copy+Replace”. In the variable property, set to the navigation bar. In the Form property, set to perObjectNavBar.Forms.Hlayout.

7 Right click on the content form frame and click “Copy+Paste”. In the variable property, set as perObjectNavBar.currentObject. In the Form property, set to perObjectNavBar.currentObject.Default (or the form to display).

8 Save and run. Click on Step 1 to display step1doc. Click on Step 2. At this point, step1doc has been validated. Go back to Step1 - validation errors should be visible. Go back to Step 2 - validation for step2doc is also visible.

4.5 Styling Validation

This example uses the previous Validate and Continue example with some styling.

Using the metadata above, create a .css file (if not present already).

Below is an example:

```
.CwCABody {
color:#000066;
font-family:Arial,Helvetica,sans-serif;
}
.CwCABodyError {
background-color:red;
color:#000066;
font-family:Arial,Helvetica,sans-serif;
}
.CwCABodyDown {
color:#000066;
font-family:Arial,Helvetica,sans-serif;
}
.CwCABodyOver {
color:#000066;
font-family:Arial,Helvetica,sans-serif;
}
.CwCABodyErrorDown {
background-color:red;
color:#000066;
font-family:Arial,Helvetica,sans-serif;
}
.CwCABodyErrorOver {
background-color:red;
color:#000066;
font-family:Arial,Helvetica,sans-serif;
}
```

1. In the **Methods** tab of validateContinueNavBar, create a new script method called “returnStep1Style”. Be sure to set the return type to String.

2. In the script, write code that will check if step1doc is already initialized; if it is and it is invalid, return “CwCABodyError”, otherwise, return “CwCABody”. If step1doc is null, then do not change the style (unless it is desired behaviour).

3. Follow steps 3-4 with returnStep2Style script checking from step2doc.

4. In the **Configuration** tab of validateContinueNavBar, click step1 and set its “Dynamic Style” property to returnStep1Style. Do the same for step2.

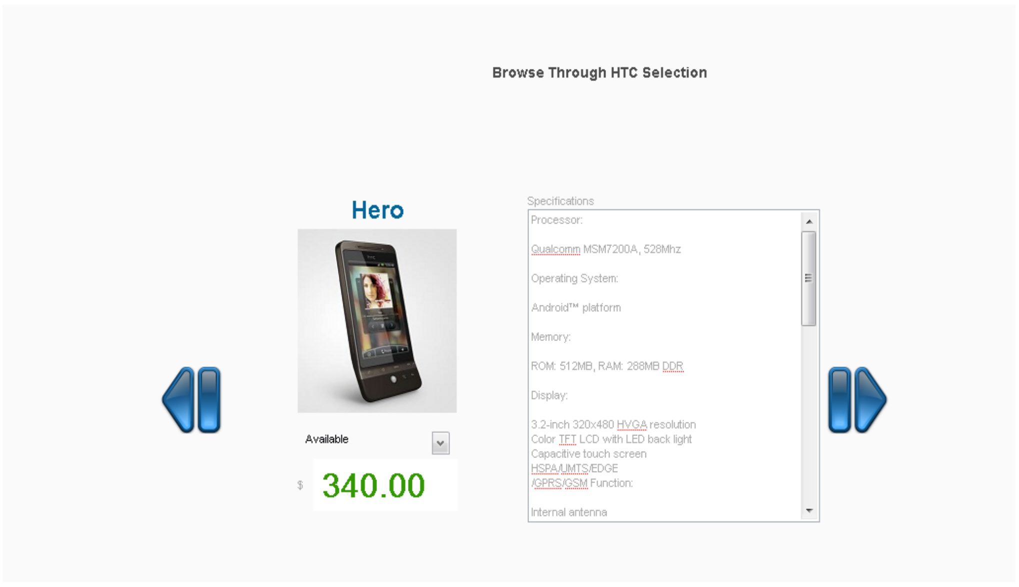
5. Save and run. In the browser, click Step 1. Do not enter any values and click Step 2. You’ll notice that Step1 now has a red outline to indicate that it is invalid. Click Step 1 again (Step 2 should now also have a red outline). Provide the necessary values to make the document valid and click Step 2 again. The red outline should now be yellow.

5 Using iterators

5.1 Per Item Iteration

5.1.1 Overview

This section explains the usage of iterators to create a complex layout.


Browse Through HTC Selection

Hero

Available

$ 340.00

Specifications

Processor:

Qualcomm MSM7200A, 528Mhz

Operating System:

Android™ platform

Memory:

ROM: 512MB, RAM: 288MB DDR

Display:

3.2-inch 320x480 HVGA resolution
Color TFT LCD with LED back light
Capacitive touch screen
HSPA/UMTS/EDGE
/GPRS/GSM Function:

Internal antenna

1 Create new metadata with the following data types:

   a availability; type String Enumeration:
   Available, Out of Stock, In Store Inquiry

   b price; type Decimal, length 5, scale 2.

2 Create a new document called “phoneDoc”.

   a Under the variables tab, add the following variables:

      i. availability; type availability

      ii. price; type price

      iii. name; type String

      iv. image; type String

      v. specs; type String

   b assign a key for this document and map to the database

3 Create a finder and set its output document to phoneDoc. Once created, navigate to the View tab of the finder and set “Auto Search” to true on the default view.

4 Under the finder user interface variable tab, add the following variables:

   a counter; type Integer
   b phoneDocArray; type phoneDoc; set array flag to true
   c currentPhoneDocArray; type phoneDoc; set array flag to true.

5. Navigate to the Methods tab of the finder user interface and override the onInit function. Below the super init call, initialize the variables:
   a. this.counter = 0;
   b. this.phoneDocArray = this.result.toArray(); //iterators support only native array
   c. this.currentPhoneDocArray = new Array();
   d. this.currentPhoneDocArray[0] = this.phoneDocArray[0]; //this array will serve as a dummy array for the iterator so that it only displays one item at a time.

6. Create a new script method called “previousClick”. In the script, add the following code:
   a. this.counter = this.counter - 1;
   b. this.currentPhoneDocArray[0] = this.phoneDocArray(this.counter);

7. Create another script method called “nextClick”. In the script, add the following code:
   a. this.counter = this.counter +1;;
   b. this.currentPhoneDocArray[0] = this.phoneDocArray(this.counter);

8. Create a new permission method called previousDisabled. Add the following code:
   a. return this.counter > 0;

9. Create a new permission method called nextDisabled. Add the following code:
   a. return this.counter < (this.phoneDocArray.length - 1);

10. Right click on the finder user interface Default form and click Override.

11. Right click on the root element of the form and click replace. From the wizard list, select Vertical Stack Layout.

12. Under the vertical stack layout, add a horizontal stack layout.
    a. Set height to 400, width 900 px.

13. For the previous button, add a vertical stack layout under the horizontal stack from step 12.
    a. Set height to 400px to use the whole height of the horizontal stack
    b. Set cell alignment to center,center

14. Under the layout from step 7, add a new menu called previous.
    a. Set click method as previousClick
    b. Set icon as /media_step_back.png
    c. Set visible as previousDisabled

15. Right click on the layout from Step 12, and add an iterator.
    a. Set variable to currentPhoneDocArray

16 Under the iterator, add a horizontal stack layout with two children:

  a vertical stack layout with children:

    i. label; set variable to name; change font property if desired

    ii. image; set variable to image

    iii. select field; set variable to availability

    iv. price; set variable to price; set label to “$” (or set dynamic label style to display the appropriate currency)

  b text area; set variable property to specs.

    i. Set editable to false

    ii. height/width to 300px

    iii. Label to Specifications

17 Under the layout from step 7, add a new menu called next.
  a Set click method as nextClick
  b Set icon as /media_step_forward.png
  c Set visible as nextDisabled

18 Create dummy data for this finder.

19 Create a new application displaying the finder created above. Run the framework. You should be able to go through each phone type in your database by clicking the previous and next button.

6 Embedding metadata objects

6.1 Simple embedded documents

This section will take you step by step into creating and displaying two meta data objects on one screen by using form frames.

1 Create a new metadata. In this metadata, create a new namespace and two documents (addressDoc, customerInfoDoc).

2 Create three new data types:
  a maritalStatus; String; Enumeration: Single, Married, Other

    i. Beside “Element properties”, select in the dropdown list Select Field.

    ii. Set a label for this data type.

b title: String; Enumeration: Ms, Miss, Mrs, Mr

  i. Beside “Element properties”, select in the dropdown list Select Field.

  ii. Set a label for this data type.

c phoneNumber: Integer

  i. Beside “Element properties”, select in the dropdown list TextField.

  ii. Set a label for this data type.

  iii. In the Display Format, set the value to \([1-9]{3}\) - [1-9][0-9{2} - [0-9]{4}

d Add more data types, if necessary

3 Navigate to the addressDoc Variables tab.

  a Add variables: (suggested)

    i. Address1
    ii. Address2
    iii. postal code
    iv. city
    v. province; type province (enumeration)
    vi. country; type country (enumeration)

  b Right click on the Default form of the addressDoc user interface (in the metadata tree left side of the designer), and select “Override”.

    i. In the form tree panel, right click **Default** and select **Add**.
    ii. An add element wizard should be presented on the screen. All forms can have only one top element, therefore it is recommended that simple form designs contain layouts as the top element to allow elements as children.
    iii. For the simplest form, a grid layout will suffice. For the purpose of this example, start with a Vertical Stack Layout for modification later on.
    iv. Right click on the vertical stack layout and select add. Select a grid layout. Click **Finish**.
    v. Right click on the grid layout, select Add, then select **Variable** under **Input Elements** folder. Click **Next**. A list of variables are presented which belong to the addressDoc. Select the variables added in step 3a. Click **Finish**.

4 Follow step 3 for customerDocInfo using the appropriate variables.

5 To display the two documents in one form, create a top level user interface:

a Right click on the namespace node in the designer metadata tree. Select **New** > **Presentation** > **User Interface**. Provide the information required (use “embeddFrameUI” as name) and click **Finish**.

b Navigate to embeddedDocApp’s **Variables** tab.

   i. Click on the add sign. In the popup, start typing addressDoc. The list should narrow down to the document and its user interface created in step 3. Select the .UserInterface item.

   ii. Add another variable of type customerInfoDoc.UserInterface.

   iii. For each newly added variable, navigate to its **Methods** tab. Right click on the tree node EmbeddedFramesUI under the **Methods** tab and select **New Intialization**.

6 Initialize each user interface variable by following the sample script: “return new namespaceName.documentName.UserInterface(null, this);”

a For example, return new EmbeddedDocNamespace.addressDoc.UserInterface(null, this);

b Override the Default form. Add a Horizontal Stack Layout (or Vertical Stack Layout).

   iv. Under the layout, add two form frames (found in Misc Controls).

   v. Click on the form frame for the addressDoc. To the right of the element tree is the element properties list.

   vi. Set the Variable property to the addressDoc variable created in step 5bi.

   vii. By default, once the variable is set, the Form property will be set to the default form of that user interface. This can be modified to point to another form within addressDoc user interface if there are others available.

   viii. Follow step ii to iv for the customerInfoDoc form frame.

7 To display the top level user interface, create an application user interface:

a Right click the namespace node in the designer metadata tree. Select **New** > **Presentation** > **User Interface**.

- Click **Next**. In the Extends property, set the value to com.conceptwave.system.Application.

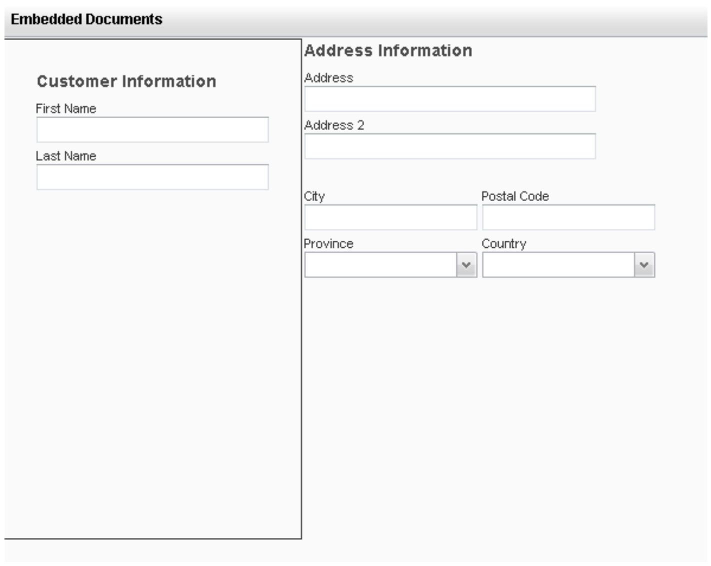
b Navigate to the **Menu** tab. Add a new user action method by right clicking the tree node.

- Set the Return property to EmbeddedFramesUI and select Default as the form.

c Override the Menu form of your application. Right click on HorizontalLayoutFormMenuItems and add a new menu.

d Set a label for the menu item and set its click method to the newly created user action method.

e Run framework. When this menu is clicked, the two documents created will be shown side by side.



6.2 Embedding Finders

This section will take you step by step into displaying a finder user interface with other metadata objects.

*For this sample, we will reuse the metadata created in the above sample.

1 Map the customerInfoDoc to a database. This document will be used to create a finder.

2 Right click on the namespace node and select **New** > **Orders and Finders** > **Document Finder**. (use customerInfoFinder as name)

3 Set the output document as customerInfoDoc. Click **Finish**.

4 Navigate to EmbeddedFrameUI. Add a new variable of type customerInfoFinder.UserInterface

5 Create an initialization value for this variable. (for example, return new EmbeddedDoc.customerInfoFinder.UserInterface(null, this))

6 Create a new form by right clicking on the EmbeddedFrameUI tree node. Call it finderDocForm.

7 Add a Vertical Stack Layout. Under this layout, add a horizontal stack element.

8 Navigate to the Default form. Right click on the HorizontalStackElement and select copy.

9 Navigate back to the finderDocForm and right click on VerticalStackElement and select paste.

10 Add another form frame under VerticalStackElement called customerFinderFrame.
   - a Set the variable property to the customerFinder variable created in step 4.
   - b Change the form property to point to customerFinder.Forms.Result instead of customerFinder.Forms.Default. This will ensure that only the table portion of the finder is displayed.
   - c Navigate to the application’s Menu tab. Add a new user action method by right clicking the tree node.
     - i. Set the Return property to EmbeddedFramesUI and select finderDocForm as the form.
   - d Navigate to the Menu form. Right click on HorizontalLayoutFormMenuItems and add a new menu.
   - e Set a label for the menu item and set its click method to the newly created user action method.
   - f Run framework. When this menu is clicked, the two documents will be shown side by side and the finder table element will be displayed directly below the documents.

6.3 Controlling Form Frame display using scripts

6.3.1 Overview

This section will take you step by step into displaying a finder user interface form controlled by scripting.

*For this sample, we will reuse the metadata created in the above sample.

1 Navigate back to EmbeddedFramesUI. Create a new variable of type String (name it finderFormName). Create an initialization for this variable (eg. return “Default”;)

2 For this example, we will create a new form under EmbeddedFramesUI but will reuse the elements create under the finderDocForm.

   - a Navigate to finderDocForm. Right click on VerticalStackElement and copy.
   - b Create a new form called finderDynamicForm. Override the Default form and paste the elements that were copied.
   - c Change the Form property for the customerFinderFrame to finderFormName variable in Step 1.

3 Navigate to your application, Methods tab. Create a new user action method with return type set to EmbeddedFramesUI and form “finderDocForm”. This method will display the finder with default form (default value of variable finderFormName).

4 Create another user action method with return type set to EmbeddedFramesUI and form “finderDocForm”.

5 In the script portion of the user action method:
   a Add a new parameter of type EmbeddedFramesUI called framesUI. By default, the product will pass in a newly initialized EmbeddedFramesUI object (as specified in the return type).
   b Modify the finderFormName variable via script (framesUI.finderFormName = “Result”)

6 Navigate to the application Menu form. Create two new forms for each user action method created.

7 Run framework. Click each menu to see the difference. With default form displayed, you will see a section stack containing the result form. If the finder were to have a search form, it would be displayed within this section stack. With the scripted menu, only the table element is shown for the finder.

7 Creating a high performance UI

7.1 Building Dynamic Forms

Form frames and section stacks give a good dynamic behavior and ability to reuse existing forms and form pieces. But from the other side every additional form frame and section stack, especially in combination with visible permissions increases page rendering, loading, and refreshing time. In some cases, it can cause your browser to run slowly or crash.

Here are some tips for avoiding or minimizing these problems.

1 When choosing a form dynamically, use String variable for Form property.

Case: When you need to choose a form of the certain User Interface depending on some criteria.

The first idea that comes to mind is to create a VLayout and put 2-5 form frames with “Visible” permissions. Or create a “mutex” section stack with several sections and dynamically set desired section visible and hide others.

Better: Create one form frame with Form property set to a dynamic variable or method that returns the current form name. Change the variable depending on criteria. Form variable can be null, then no form will be displayed.

Why: to improve page performance, try to minimize number of Form Frames and Section Stacks.

2 **Instead of using one dynamic “Default” form, create and use static custom forms.**

Case: This issue is related to the first one. Quite often we use same User Interface or Document as a part of other User Interfaces and we might need to change look and feel depending on where it’s displayed.

The usual approach is to make the Default form change look and feel dynamically.

Better: Create different forms for different cases. When adding a Form Frame for the User Interface, select the appropriate Form at Design time.

As an example, if you have finder and you need to display only result (table), use “Result” form directly instead of overriding finder’s “Default” form and hiding search form and other sections.

Variable can be used for form names under iterator as well.

Using “Default” form makes sense only if it’s only one form or the form name is unknown at Design time (when for ex, iterating different types of UI).

3 **Be conscious when splitting forms into small pieces.**

It is a usual developer’s practice to split forms into small blocks and reuse them to build page dynamically. But keep in mind that this increases rendering and building page at runtime. It’s not worth to reuse tiny parts, like “header”, “border”, etc. The form in “one piece” might be displayed 10 times faster than the form that consists of 10 small forms.

4 **To reduce form elements rendering and showing time:**

- Avoid adding extra V- and HLayouts when it’s not necessarily. For simple forms GridLayout works better. For complicated forms use V- and HStack elements.

- Set fixed width and height for form frames, layouts, stacks and dialogs whenever possible, it will reduce drawing time significantly.

- From a performance perspective, it’s better to create several different forms instead of using one form with “Visible” permissions on elements.

Keep in mind that every time it renders the entire form with all elements and only then hides elements that are invisible.

5 **Avoid showing complicated forms in dialogs.**

In some cases it would be faster to display the form right on the page and then go back to the original flow.

It is a known issue that when browser displays a modal dialog with complicated structure (with many nested form frames, layouts and big number of elements) it takes much longer then to display the same form right on the page.

One of the reasons is that the dialog is not a separate window, it is still a part of the current page and the browser applies a “mask” to all page elements to separate “active” dialog controls from “inactive” page controls every time on showing or reacting on user actions.

6 **TabSet vs. Navigation Bar**.

Using TabSet with heavy loaded tabs can be a performance issue. All forms from tabs, including hidden/not selected, are created at the page loading. From the performance perspective it is the same as having all tab forms on the same page.

It is recommended to use Navigation Bar which can look like tabs for heavy tab pages, when you have one UserInterface controller per one tab. As advantage, you will also have some ready functionality: on tab changed notification, auto validation and displaying error status on the tab icon.

And vice versa, for light forms in tabs, when you do not need “on tab changed” notification, it would be faster to use TabSet because all forms are created right away, so changing tabs does not take any time and does not cause round trip to the server.

7.2 V/HStack vs. V/HLayout

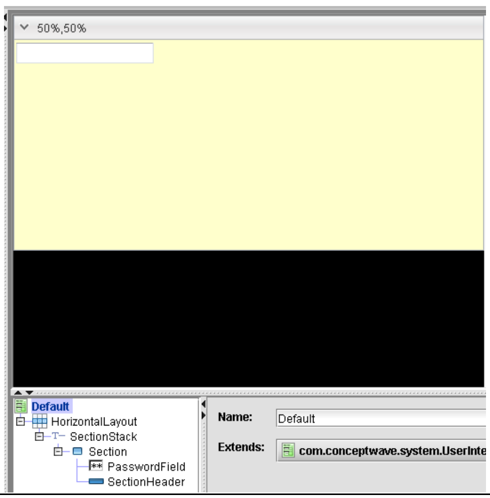
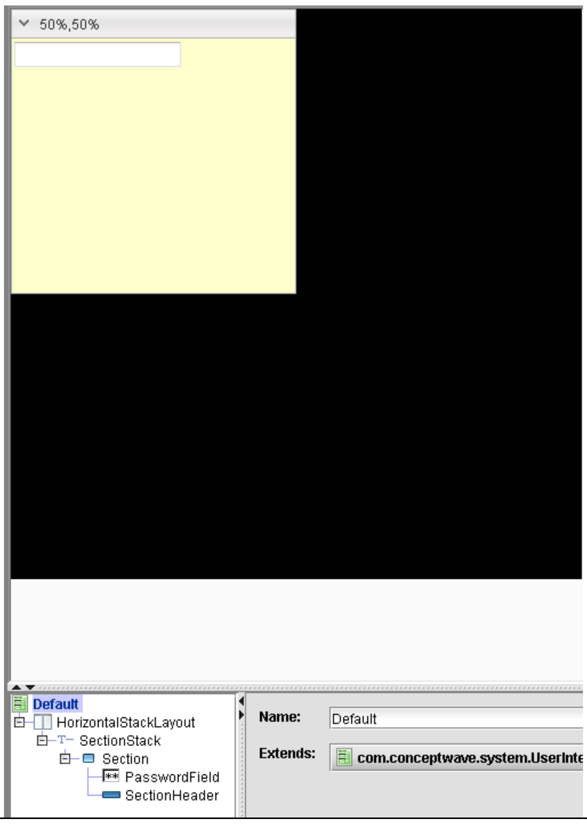
It is recommended to use Vertical and Horizontal Stacks rather than Vertical and Horizontal Layout elements.

VStack and HStack elements are quite similar to Vertical Layout and Horizontal Layout elements in terms of overall flow of its children. Horizontal layout and horizontal stack will draw its children horizontally, one after another and vertical layout and vertical stack will draw its children vertically. The only major difference between the two types of layouts is the handling of child sizing in percentage. Vertical/Horizontal layouts will adjust its children to fill the available space when needed, whereas the stack layouts will always respect the width/height values set on the children.

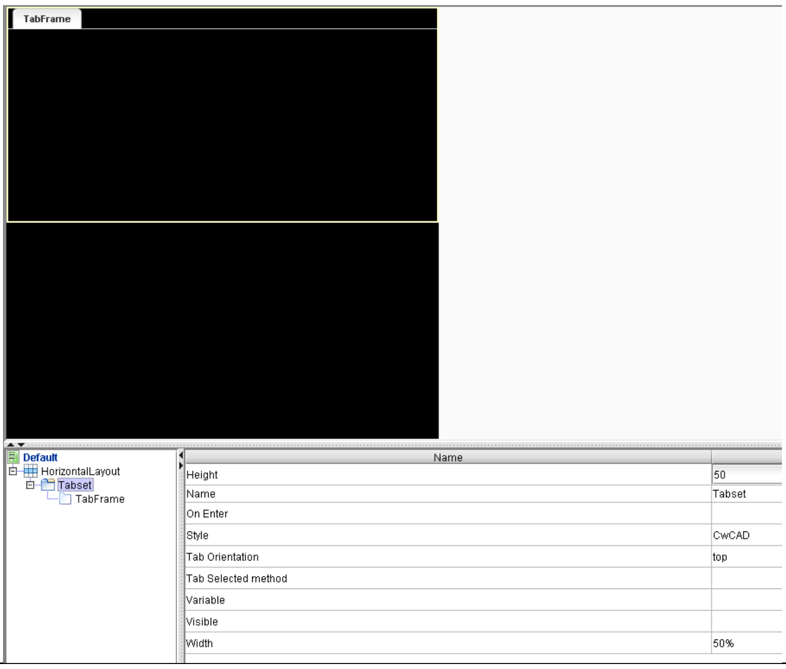
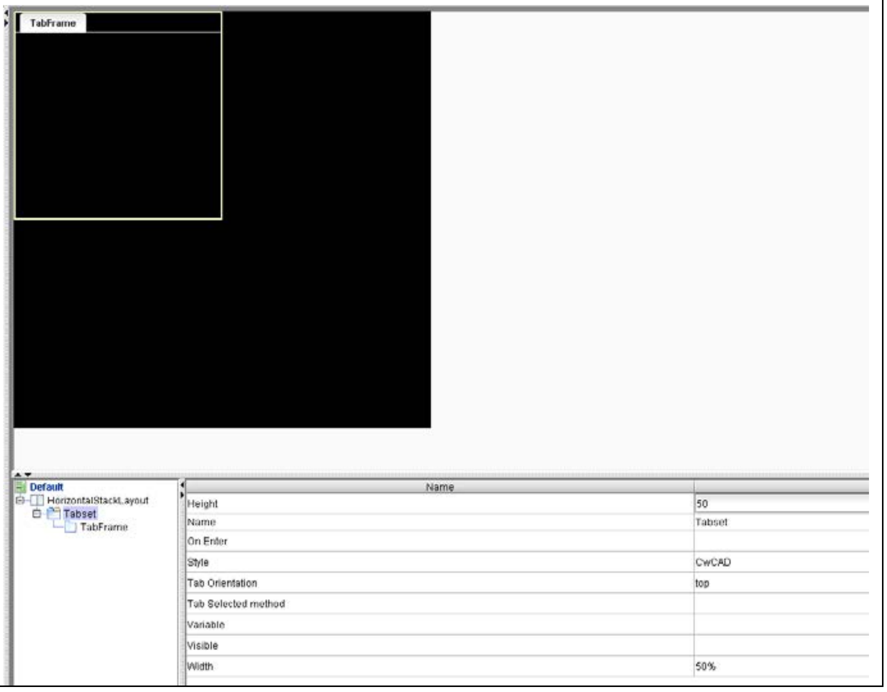
7.2.1 Comparison

Below is a comparison of the two layouts.

SectionStack width/height 50%

HLayout	 50%,50% Default HorizontalLayout SectionStack Section PasswordField SectionHeader Name: Default Extends: com.conceptwave.system.UserInte
HStack	 50%,50% Default HorizontalStackLayout SectionStack Section PasswordField SectionHeader Name: Default Extends: com.conceptwave.system.UserInte

Tabset width/height 50%

HLayout	
HStack	

HLayout screenshot:

```
Default
HorizontalLayout
  Tabset
    TabFrame
```

Name	
Height	50
Name	Tabset
On Enter	
Style	CwCAD
Tab Orientation	top
Tab Selected method	
Variable	
Visible	
Width	50%

HStack screenshot:

```
Default
HorizontalStackLayout
  Tabset
    TabFrame
```

Name	
Height	50
Name	Tabset
On Enter	
Style	CwCAD
Tab Orientation	top
Tab Selected method	
Variable	
Visible	
Width	50%

7.2.2 Default Behaviour

Vertical/Horizontal Layout	**Vertical/Horizontal Stack**
Height/Width in Percentage	
Vertical and horizontal layouts, if presented with only one immediate child (which has height/width in percentage), are rendered with a layout spacer as its second child to fill the rest of the available space.	Vertical and horizontal stacks are rendered as it is designed in the metadata. This means there are no added layout spacers.
Alignment	
For every layout alignment, there is at least one layout spacer created within the overall layout to achieve property alignment. For example, For a horizontal layout aligned “center, center” with one child, the following are created: horizontal parent layoutspacer as first child child element layoutspacer as last child **If the child element is a horizontal element with alignment “center,center” another two layout spacers are created within.**	For every layout alignment, no extra layout spacers are created. For example, For a horizontal layout aligned “center, center” with one child, the following are created: horizontal parent child element In more complex user interfaces, this behaviour is much more desirable as there are less elements on the screen.

8 Dialog samples

8.1 Dialog with frozen bottom menu [dialog.DialogExt]

This sample shows how to make 5.2 Dialog object follow 4.2 style:

Menu is frozen at the bottom.

Menu style is same as application menu.

1 Create a new namespace: “dialog”.

2 Create new UserInterface object under the namespace: “DialogExt”.

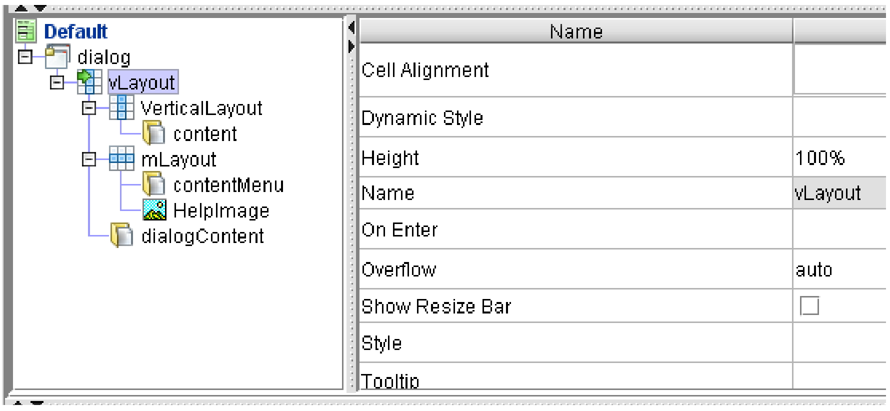
3 Set Override property: com.conceptwave.system.Dialog.

4 Override “Default” form, right click at the “vLayout” element and do “Copy And Replace” action.

5 Then change the layout for the dialog form frames: create an additional “VerticalLayout” for the “content” form frame, move “content” form frame under “VerticalLayout”.

6 Set layouts properties:

```
“vLayout”: width – 100%, height – 100%, overflow – auto,
   “VerticalLayout”:  width – 100%, height – 100%, overflow
– auto,
        “content” for frame: width – 100%, height – 100%,
overflow – visible,
    “mLayout“: width – 100%, height – none (empty), overflow
– <Default>,
        “contentMenu” form frame: width – 100%, height –
1px, overflow – visible.
```



```
Default
⊟ dialog
   ⊟ vLayout
      ⊟ VerticalLayout
            content
      ⊟ mLayout
            contentMenu
            HelpImage
         dialogContent
```

Name	
Cell Alignment	
Dynamic Style	
Height	100%
Name	vLayout
On Enter	
Overflow	auto
Show Resize Bar	☐
Style	
Tooltip	

7 Change style of the “mLayout” to “button” style.

Note: If you are unable to select “button” style in the combo box, you can change it manually right in XML file. Save and close the project; find ui_DialogExt.xml into your metadata directory; open it and add <style> tag for the mLayout element:

```
 <element name="mLayout" type="elhlt">
     <style>button</style>
     <width>100%</width>
           <elementList>
              ……
```

9 Navigation tree samples

9.1 Tree node with dynamic children [sample.tree.Dynamic]

This section will show how to use Finder Tree Node to have dynamic child nodes returned by the finder.

Our goal is to build a navigation tree with the folder node “Customers” and a list of customers under the folder. The customer node is going to have dynamic label and icon and the editable detail view. The folder node will also have search/filter functionality, “Add customer” and “Delete” actions.

1 Creating “sample.tree.CustomerFinder”

   Start by creating a finder for the dynamic node.

   a Create an Enumeration data type “sample.tree.CustomerType” with some values, for ex.:

      - CM (Commercial);
      - PA (Partner);
      - PR (Private).

   b Create an output document “sample.tree.customer” with fields like “firstName”, “lastName”, “type” (type of sample.tree.CustomerType), generated document ID, etc. Map the document to DB. Overwrite and fill the Default form of the document’s UserInterface with fields bound to the variables.

   c Create an input (search) document “sample.tree.customerSearch” with same fields: firstName”, “lastName” and “type”. Fill the Default form with fields.

   d Create a document finder “sample.tree.CustomerFinder” and select “sample.tree.customerSearch” as an input document and “sample.tree.customer” as an output. Define search criteria by mapping variables with same names:”firstName”, “lastName”, “type”.

2 Creating “sample.tree.Dynamic” navigation tree

   The next step is to create a navigation tree and use Customer finder to fetch customer nodes.

   1 Add a new Navigation Tree “sample.tree.Dynamic”.

   2 Right-click at the Navigation tree node and add a new Finder Node “Customers”. Specify node’s properties:

      Object: sample.tree.CustomerFinder

      Detail: sample.tree.CustomerFinder.UserInterface

Form: Search

Auto Expand: selected

The “Object” property defines the type of the “model” variable for the tree node and can be referenced in scripts as “this.model”. For the “Customers” tree node the model will be the customer finder.

If we specify the object’s UserInterface as Detail, then the model will be passed to the UserInterface constructor so the Detail view will display that particular finder detail. Thus, the model of the tree node and the model of the detail view will be the same object (instance).

3 When adding the finder node, it automatically creates “FinderChild” node for the output document. That node will be replicated for each finder result document. The Object property is set to the “sampl.tree.customer” document automatically.

Edit the “FinderChild” node properties to specify document’s Detail view:

Detail: sample.tree.customer.UserInterface

Form: Default

4 Go to the “sample.tree.Dynamic” navigation tree properties and select “Exclude” checkbox to exclude the root node.

Excluding a node means that children of the node will be displayed at the parent’s level. Because we are excluding the root node, “Customers” finder node will be displayed in browser right at the top level as the tree root.

5 Dynamic Label and Icon

If the model of the tree node is a document or an order item, the default implementation of the “onNodeVisualKey” method returns the visual key of the model. In other cases or if the visual key is empty, the default static label of the node is displayed.

```
Override the “onNodeVisualKey” and “onNodeIcon” methods
for the “FinderChild” to animate the customer node:

// FinderChild’s onNodeVisualKey: first + last name

if (this.model.firstName == null && this.model.lastName
== null)

  return "New Customer";

return this.model.firstName + " " + this.model.lastName;

// FinderChild’s onNodeIcon: return different icon
depending on the customer type

var icon = "/cwt/images/24/user3.png";

if (this.model.type == "CM")

  icon = "/cwt/images/24/user1_into.png";
```

```
else if (this.model.type == "PR")
  icon = "/cwfv/user_preferences.png";
// Add error.png overlay if the document has errors
if (this.model.hasErrorStatus)
  icon += ",/cwf/ov/error.png(bottom-right)";
return icon;
```

3 Displaying the Navigation Tree

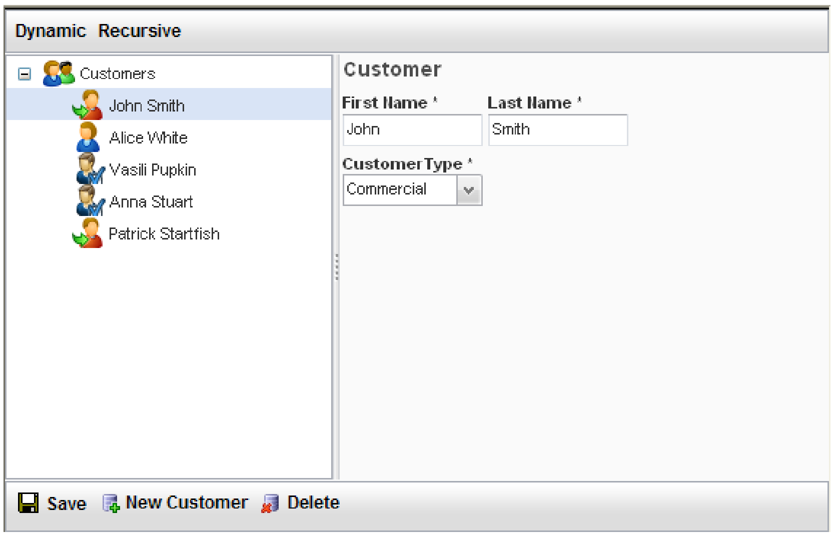
As any other metadata object like top-level User Interface, document or finder, the navigation tree can be created by User Action or by scripts.

Create an Application UI sample.tree.app and create a new User Action method where specify sample.tree.Dynamic navigation tree as Object and select “Default” form to display.

Add a menu item to the application’s Menu form and select the new action as “Click method”.

Use one of the following options to create the navigation tree through scripts:

```
var navTree = new NavTree(“sample.tree.Dynamic”); // or
navTree = new sample.tree.Dynamic();
return navTree;
```



Customer

First Name *	Last Name *
John	Smith

Customer Type *: Commercial

sample.tree.Dynamic Navigation Tree at runtime

4 Tree Node Menu

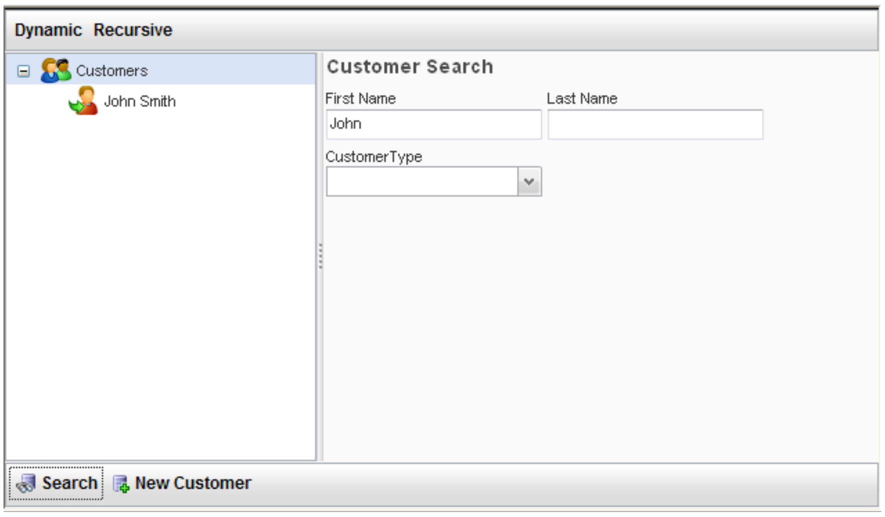
As you probably already know, when Application UI displays an object as content, it also shows the object’s Menu at the bottom of the page.

The default “Menu” form of the Navigation tree is simple: it has one form frame that points to the Menu form of the currently selected node in the tree. The node’s Menu form also has “childMenu” form frame pointing to the Detail’s UI Menu form.

In the picture above “Save” menu item comes from the sample.tree.customer.UserInterface. Other actions “New Customer” and “Delete” are created under the FinderChild’s Menu form.

If you select the “Customers” node (next picture) you will see that Menu changes. Now it shows the finder’s default Menu with “Search” and other actions.

Type search criteria and try “Search” action. The list of customers in the tree changes according to the search results. Every time when search has been performed on the finder, as well as add, update or delete operations, the “Customers” finder node receives notifications and reflects finder changes.



First Name: John
Last Name:
CustomerType:

Running Search action in CustomerFinder filters ChildFinder nodes

Note: Default finder menu “Add”, “Copy”, “Delete”, etc. actions are intended to be used with the “Default” finder’s form and not working properly with the navigation finder tree node when displaying “Search” as Detail Form. It is recommended to override the Menu of the Finder Node, remove the default “childMenu” Form Frame and build your own menu for the node.

For example, to implement “Search” action at the “Customers” node level, you can access the Detail UI via “modelUI” variable:

```
// Customers node: “Search” action
this.modelUI.searchAction();
```

5 “New Customer” action

Now we would like to add new customers to the list.

a Create a new “newCustomer” User Action method of the “Customers” tree node:

```
// Customers: newCustomer action
// get finder UI:
var finderUI = this.onNodeDetail(); // *
if (finderUI != null) {


// add new document to finder
var addParam = new
Document(this.model.selectOutputDocumentMetadataName);
this.model.metadata.operations.add.invoke(addParam,
finderUI.invokeContext);
// NOTE: When the new document is added by "add"
operation, the finder tree node
// is being notified and adds a new node to the tree
automatically


// Find the new node by ID: **
var newID = this.tableDoc.id + "/FinderChild#" +
addParam.cwDocId;
var newNode = this.findNode(newID);
if (newNode != null) {
  // "displayNode" function takes care of selecting the
node in the tree,
  // expanding all parents and displaying detail view
  this.getRoot().displayNode(newNode);
}
}
```

* We called “onNodeDetail” at the beginning to initialize the “modelUI”. This variable has lazy initialization so the Detail UI is created when it is selected in the tree for the first time.

** When we are adding a new document to the finder, it actually adds a copy of that document to the result list. Thus, we cannot use the “addParam” object to find a node by model (findNodeByModel method). As a workaround, calculate the new node ID and use it to find the new node.

The ID of the finder child node consists of the parent node ID + “/” + node name + “#” + document ID. In this case the ID would be “Dynamic/Customers/FinderChild#123456”.

b Add a new Menu Item to the “Menu” form of the “Customers” node. Select the “newCustomer” action as Click method.

c If we also want to have this action on the customer node level, we need to add the same user action “newCustomer” and the same menu item to the “FinderChild” tree node’s Menu.

But in the user action code we can just call the parent node method:

```
// FinderChild: newCustomer action
this.parentNode.newCustomer();
```

9.2 Recursive tree nodes [sample.tree.Recursive]

This sample shows how to create recursive trees when each child node repeats the parent node structure. This sample has the “node” document which can have dependent child “node” documents, etc. recursively.

1 Creating sample.tree.childNodeFinder

   This finder looks up the node’s children by their parent ID.

   a Create a document for the node: tree.sample.node with variables: name, description, parentID (type of cwf.String64) and generated document ID: cwDocId. Map the document to DB.

     Fill the “Default” form with “Name” and “Description” fields.

     Override the visual key method (cwOnDocVisualKey) for the document:

     ```
     // cwOnDocVisualKey
     return this.name == null ? "***" : this.name;
     ```

   b Create a search document tree.sample.nodeSearch with only one variable “nodeID” (type of cwf.String64).

   c Create a document finder tree.sample. childNodeFinder with “tree.sample.nodeSearch” as Input and “tree.sample.node” as Output.

     Define search criteria: map nodeSearch.nodeID = node.parentID.

   d We also need a conversion map that converts the parent node document to the search document for children.

     Create a “sample.tree.node2nodeSearch” conversion map with the following properties:

     Source: sample.tree.node

     Target: sample.tree.nodeSearch

     Map fields: sample.tree.node.cwDocId -> sample.tree.nodeSearch.nodeID

2 Creating sample.tree.Recursive Navigation Tree

The recursion can be done with only 3 nodes: a finder node to look up for child nodes, the FinderChild node for result documents and a recursive node to implement the recursion.

a Create a new Navigation Tree object “sample.tree.Recursive” with Object property set to ”sample.tree.node” document.

  This is our root node, it will be used as a parent node for children with parent ID = <root_id>.

  Note: The parentID cannot be empty, so for root nodes we need not empty <root_id>, otherwise for the “nodeID = null” condition the childNodeFinder will return all existing sample.tree.node documents with any parent ID (null or not null).

  The root document does not need to be stored in DB; it can be virtual. To initialize the cwDocId of the root document we override onInit() method of the navigation tree root node:

```
// sample.tree.Recursive root: onInit()
this.model.cwDocId = "1"; // “1” is the root ID
this.model.name = "Root Node"; // Visual key for the root
document
this.cw$super_onInit();   // then call super method
```

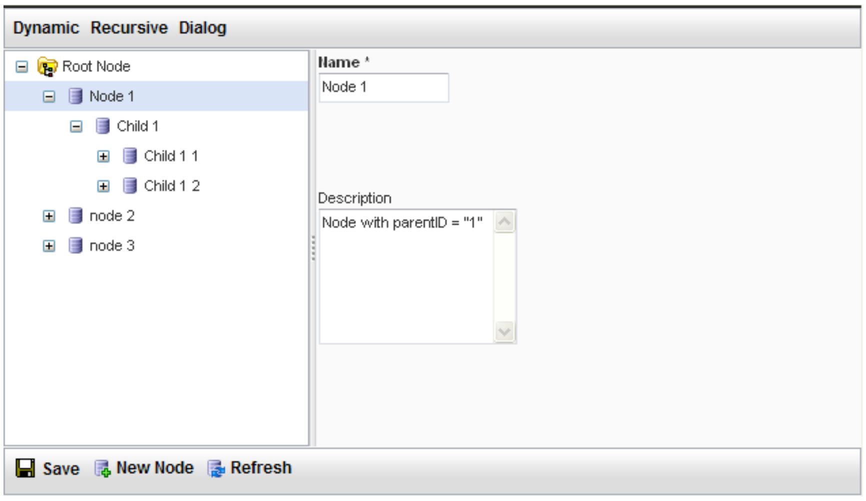
b Add a finder node “NodeFinder” and select sample.tree.childNodeFinder as “Object” property. Choose “sample.tree.node2nodeSearch” as “Search conversion map”. Select “Exclude” check box: this will be a virtual node, and will not be visible at runtime.

c The FinderChild node under the “NodeFinder” is added automatically. Edit the node’s properties:

  Detail UI: sample.tree.node.UserInterface

  Form: Default

d Under the “FinderChild” node add a new Recursive Node “recursiveNode” and select Recursive item: sample.tree.Recursive.NodeFinder.



Recursive tree at runtime

How it works at runtime starting from the root:

a On expand event of the root node, it creates a “NodeFinder” instance with empty search document. On “NodeFinder” initialization, the search conversion map passes the cwDocId of the root document to the “nodeID” of the search document.

b Because the “NodeFinder” node is excluded, its children will be loaded immediately and displayed right under the root node. The “FnderChild” node will be replicated for each found child sample.tree.node document and will have dynamic label equal to the visual key of the document.

c Recursion: at runtime the metadata of the “recursiveNode” is being replaced with the metadata of the “Recustive Item” node.

d In our case, the runtime child under the “FinderChild” will be the “NodeFinder”. So when we expanding a “FinderChild” node, it creates a new instance of the “NodeFinder” node and once again, it applies the search conversion map passing FinderChild’s “cwDocId” to “nodeID” of the search document to look up for children of the next level.

3 “New Node” action

The “New node” action is common for all tree nodes, so it is more convenient to have it on the “Menu” for of the tree.

a Create “newTreeNode” User Action method on the root node level:

```
// get selection to retreive parent ID
var parentNode = this.getSingleSelection();
if (parentNode == null)
  parentNode = this;

// Create new "node" document:
var newNodeDoc = new sample.tree.node();
newNodeDoc.parentID = parentNode.model.cwDocId;
newNodeDoc.save();
```

```
// Add the new node to the child finder
if(!parentNode.loaded) {
  //first load children if not loaded yet
  parentNode.loadChildren();
}
// Create new instance of "FinderChild" tree node, pass
new model as second parameter
var newTreeNode =
parentNode.childNodes[0].newNode("FinderChild",
newNodeDoc);
// Display the new node
if (newTreeNode != null) {
  this.displayNode(newTreeNode);
}
```

b Override “Menu” for of the root node (navigation tree forms), add a new menu item “New Node” and select “newTreeNode” user action as “Click method”.

4 “Refresh” action

Children of the finder node are being automatically refreshed on “search” action of the finder.

But you might need to refresh the node and its children manually. Use the following method:

```
treeNode.refreshNode(reloadChildren, keepExpanded);
```

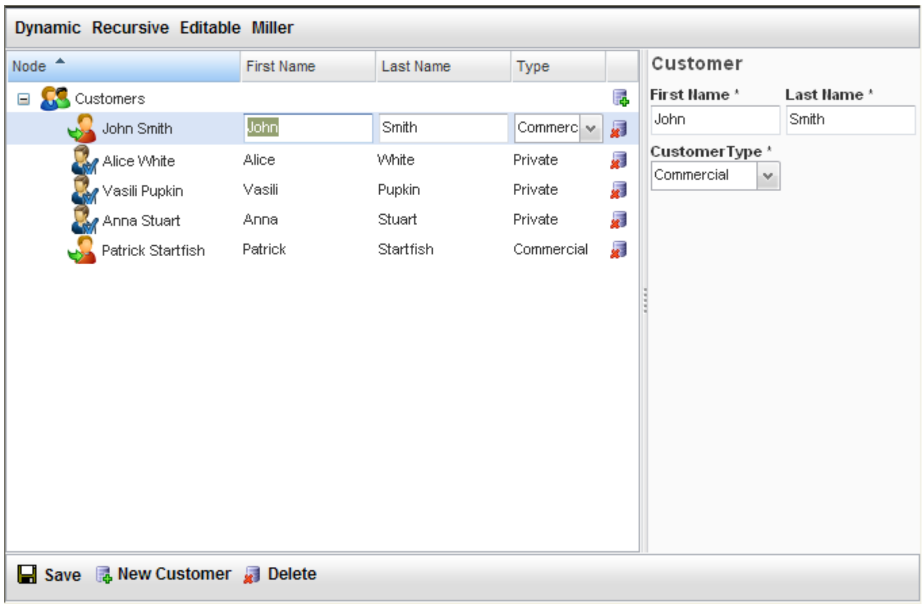
Parameters:

`reloadChildren` (true if not specified) - If true, children of the node will be reload and recreated;

`keepExpanded` (true if not specified) - If true to re-expand the node's children that was expanded before refreshing, if false child nodes will be expanded according to “Auto expand” property.

9.3 Editable Tree [sample.tree.EditableTree]

“sample.tree.EditableTree” sample navigation tree is an editable version of “sample.tree.Dynamic” tree. The editable tree will allow editing first and last name and type of the customer. Plus it will have an icon action column: “Add” for “Customers” node and “Delete” for customer leafs.



Node	First Name	Last Name	Type
Customers			
John Smith	John	Smith	Commerc
Alice White	Alice	White	Private
Vasili Pupkin	Vasili	Pupkin	Private
Anna Stuart	Anna	Stuart	Private
Patrick Startfish	Patrick	Startfish	Commercial

Customer

First Name *	Last Name *
John	Smith

CustomerType *: Commercial

sample.tree.EditableTree at runtime

To make a navigation tree editable, you need to do the following:

a Set “Editable” property of the tree form element to TRUE or any permission method.

b Add editable columns (text, select, check box, etc.) to the form tree element.
Note: Label column still will not be editable, even if the tree is editable.

c Extend default “Table Document”, add leafs for editable column values and then bind columns to those leafs (will be explained late in this section).

d Implement “onNodeEdited” method for each tree node to apply modified values to the tree node model.

Table Document

Any navigation tree has the “Table Document” property. By default it is set the system document type: “com.conceptwave.system.TreeDocument”. This default document has variables to keep unique information about each tree node at runtime: id, label, image, and states: visible, expanded, etc.

This document is created in “initNode” method of the tree node and stored in “tableDoc” variable.

The “Table Document” is also used as common (interface) type to bind custom tree columns for different tree node models.

navTree Variable

If you look at the “Default” form’s “tree” element, you will see that “Variable” property points to the “navTree” variable of the navigation tree root. This “Array” variable serves as the data source for the tree form element and it always has type of the “Table Document”.

“navTree” is special scriptable **TreeDataSource** object that just has “Array” definition but actually is not an array. It does not support getting item by index, but has other useful properties like “selected”, “selectedCol”, “hover”, etc. (See scriptable objects API documentation.)

Column Variable

When adding a new column you can bind it to any variable of the “navTree” or any method of the root node returning String. Editable column though can be bond only to “navTree” variables.

By default the “tree” element has only one label column “Node1” bound to “navTree.label” variable. The first column is always a column with node image and “expand” icon. It can be removed and replaced with any other column type or be bound to other “navTree” leaf.

Table Conversion Map

“Table Conversion Map” is used to convert the tree node Object to the Table Document. Conversion map is applied only for document and finder models. In case of finder it gets the search document as the target document. The map is applied on node initialization and every time on node update (see “applyTableDocMap” method).

This property is optional and can be set per node. All default “tableDoc” leafs are initialized or set by tree node scripts or internally. For example, “id” is initialized with “onNodeID” method, “label” and “image” with “onNodeVisualKey” and “onNodeIcon” respectively. For details, see “initNode” and “updateTableDoc” methods.

To provide custom conversion, override the “updateTableDoc” method and initialize “tableDoc” leafs with model values manually.

1 Creating tree

   Let’s start from creating a new navigation tree:

   a If you already have sample.tree.Dynamic tree in your metadata, just copy&paste the entire tree and rename to “sample.tree.EditableTree”.
   b Override the “Default” form, then find “hLayout2” element on the form and do “Copy And Replace…” action for it.
   Set width of the hLayout2 element more than 400px to create space for columns.
   c Set “Editable” property of the form “tree” element to “TRUE”.

2 Adding Editable Columns

   a Create a new document “sample.tree. EditableTreeDocument” that extends the system “com.conceptwave.system.TreeDocument”.
   b Add leafs to the document: “firstName”, “lastName” and “type” with same data types as leafs of “sample.tree.customer” document.

- c Change “Table Document” property of the navigation tree root to the “sample.tree.EditableTreeDocument”. After you change the property, the type of the “navTree” variable automatically changes to the new document type.

- d Add editable columns to the “tree” element on the “Default” form of the root node:

  - i. Text field column “firstName” with “Variable” set to navTree.firstName;
  - ii. Text field column “lastName” bound to navTree.lastName;
  - iii. Select field column “type” bound to navTree.type.

3 Table Conversion Maps

- a Create a new conversion map for the search document of the “Customer” tree node:
  Name: sampel.tree.CustomerSearch2EditableTreeDocument
  Source: sample.tree.CustomerSearch
  Targer: sample.tree.EditableTreeDocument
  Map fields: firstName, lastName, type.

- b Set “Table Conversion Map” property of the “Customers” finder node to “sampel.tree.CustomerSearch2EditableTreeDocument” map.

- c Create a new conversion map for the “FinderChild” tree node:
  Name: sampel.tree.customer2EditableTreeDocument
  Source: sample.tree.customer
  Targer: sample.tree.EditableTreeDocument
  Map fields: firstName, lastName, type.

- d Set “Table Conversion Map” property of the “FinderChild” node to “sampel.tree.customer2EditableTreeDocument” map.

4 Implementing “onNodeEdited” method

When row editing is done in browser, the browser sends a special request to the server with the edit event notification, all modified values are being stored into “tableDoc” and then the “onNodeEdited” method is being invoked for the edited tree node.

Note that “Table Conversion Map” works only one-way: from “model” to “tableDoc”. To apply edited “tableDoc” values back to the tree node “model”, you need to override “onNodeEdited” method for each tree node separately and put backward conversion there.

We can implement different logic for our tree nodes. Modifications for the “Customers” finder node row will be applied to the search filter. Customer’s row modifications will be applied and stored in the “customer” document.

5 Go to the “Customers” node and override “onNodeEdited” method:

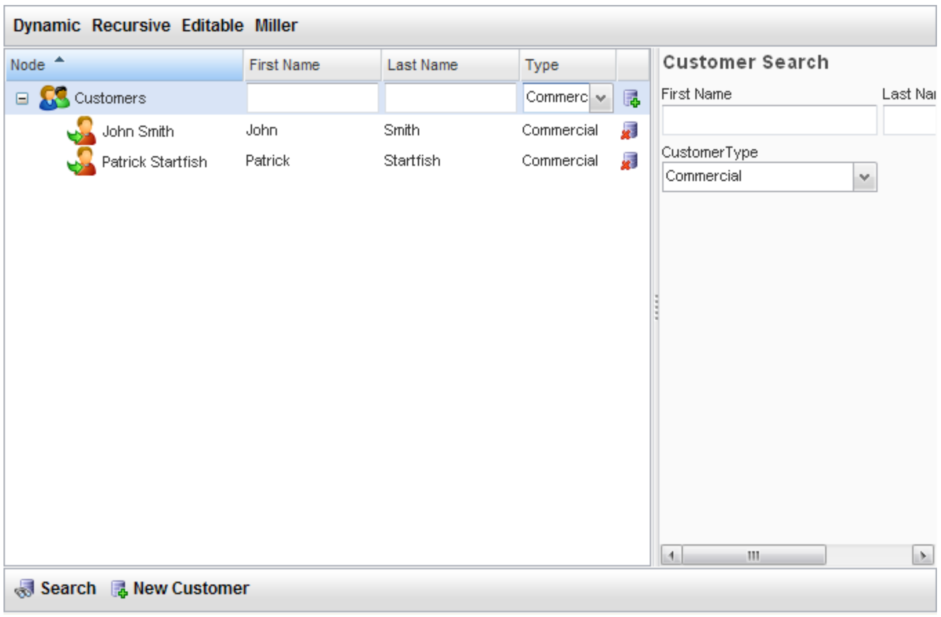
```
var changed = false;
var searchDoc = this.model.searchDocument;
```

```
if (searchDoc != null) {
  if(this.tableDoc.firstName != searchDoc.firstName) {
    searchDoc.firstName = this.tableDoc.firstName;
    changed = true;
  }
  if(this.tableDoc.lastName != searchDoc.lastName) {
    searchDoc.lastName = this.tableDoc.lastName;
    changed = true;
  }
  if(this.tableDoc.type != searchDoc.type) {
    searchDoc.type = this.tableDoc.type;
    changed = true;
  }
}
if (changed) {
  var root = this.getRoot();
  var selected = root.getSingleSelection();
  // run search action to apply filter
  this.model.search();
  // restore selection: or select finder node if selected
node is gone
  if (selected != null) {
    var newNode = root.findNode(selected.tableDoc.id);
    if (newNode == null)
      newNode = this;
    if (selected != newNode) {
      root.displayNode(newNode);
    }
  }
}
```

6 Override “onNodeEdited” method of the “FinderChild” tree node:

```
var changed = false;
if(this.tableDoc.firstName != this.model.firstName) {
  this.model.firstName = this.tableDoc.firstName;
  changed = true;
}
if(this.tableDoc.lastName != this.model.lastName) {
  this.model.lastName = this.tableDoc.lastName;
  changed = true;
}
if(this.tableDoc.type != this.model.type) {
  this.model.type = this.tableDoc.type;
  changed = true;
}
if (changed) {
  // it is better to validate modelUI:
```

```
  // it will validate the document and display errors in the
detail form
  var valid = true;
  if(this.modelUI != null && this.modelUI.onValidate()) {
    valid = this.modelUI.onValidate();
  }
  else {
    // if Detail is not set for tree node, validate the
document itself
    valid = this.model.validate(5) == null;
  }
  // save the document only if valid
  if (valid)
    this.model.save();
  // update node's label, icon, etc.
  this.updateNode(true, false, false);
}
```



Node	First Name	Last Name	Type
Customers			Commerc
John Smith	John	Smith	Commercial
Patrick Startfish	Patrick	Startfish	Commercial

First Name
Last Na
CustomerType
Commercial

Editing “Customers” node row to specify search filter

Note: If during applying modification you change node’s label, image or other “tableDoc” variables and do not call “updateNode” method, you need to mark the node “dirty”:

```
this.markDirty();
```

This will force the node and node’s children to be refreshed in the browser tree.

7 Icon Column Action

To fire icon column trigger, the table does not need to be editable. The trigger will be fired in any case.

Add an action column to the tree: for the “Customers” finder node, it is a “New Customer” action and “Delete” for customer rows.

a Add Image field under the “tree” form element.

b Create a dummy method “getActionIcon” method on the root node level.

   Select Return type: com.conceptwave.system.String

```
// This dummy action can be overriden by tree nodes
return null;
```

c Select “getActionIcon” method as “Variable” for the image column.

d Create another dummy method “actionTrigger” of the tree root node and select it as “Click method” for the Image column.

   Here comes the tricky part. We are going to create methods with the same names (and parameters) for tree nodes that should have icon and action for the Image column. For any other tree nodes that do not “override” those methods, the root “dummy” method will be invoked.

   You can implement methods only at the root level. First parameter of the methods (if defined) will be the table document of the action node.

e Create “getActionIcon” method of the “Customers” finder node:

```
return "/cwt/images/16/data_add.png";
```

f Create “actionTrigger” method of the “Customers” finder node and call “newCustomer” action:

```
// Action: new customer

this.newCustomer();
```

g Add same methods for the “FinderChild” node but for Delete action:

```
// getActionIcon:
return "/cwt/images/16/data_delete.png";


// actionTrigger:
this.deleteCustomer();
```

9.4 Data Structure UserInterface [sample.ds.testDs2.UserInterface]

The “UserInterface” child of the data structure metadata object is based on the Navigation Tree which allows browsing data structure items at runtime.

This sample shows how to build a navigation tree for data structure items and update tree when data and items are changing in the DataStructure runtime object.

1 Creating data structures

Our sample is quite complicated: beside the data type fields the master data structure (sample.ds.testDs2) will have an array of other data structures (sample.ds.testDs1), which also will have an array of documents (sample.ds.dsDoc).

a Create new “sample.ds.dsDoc” document. Add “textField” document String variable, make it mandatory. Override the “Default” form and add a text field for the “textField” variable.

Override “cwOnDocVisualKey” method of the document and return “textField” variable:

```
return this.textField == null ? "***" : this.textField;
```

b Create new “sample.ds.testDs1” data structure. Add 3 data structure items:

attrA – Data type: checked; Element: String; Mandatory: checked;

attrB – Data type: checked, Element: String;

docs – Array: checked, Element: sample.ds.dsDoc.

c Create new “sample.ds.testDs2” data structure. Add 3 data structure items:

attrDs2A – Data type: checked; Element: String; Mandatory: checked;

attrDs2B – Data type: checked, Element: Decimal;

dsArray – Array: checked, Element: sample.ds.testDs1

2 Synchronizing “UserInterface”

a Right click on the “UserInterface” child of the “testDs1” data structure and click “Sync Navigation Tree” action.

The action creates corresponding tree nodes for the “docs” item: the container node “docs” and document instance node “docsInstance” and automatically populates “Detail UI” property for the document node.

Note: This action does not create tree nodes for data type items and arrays of data types. Only document and data structure elements are created.

b Perform “Sync Navigation Tree” on the “testDs2” UserInterface.

Note: This step creates tree nodes not only for items of the master "TestDs2" data structure ("dsArray" and "dsArrayInstance"), but also goes deeper and creates "docs" and "docsInstance" nodes under "dsArrayInstance" according to the "Element" structure hierarchy.

Set "Auto Expand" properties for the "dsArray" and "saArrayInstance" tree nodes.

You can also add "Data Structure Item" tree nodes to the "UserInterface" manually. When adding a new node, bind the node by "Data Structure Path" property.

It also allowed having any other types of tree nodes under any Data Structure Item node.

Moreover, if you have a static node in any navigation tree with "Object" property set to a data structure, then you can add the Data Structure Item child nodes and bind them to the data structure elements.

c Generating forms for data type items

We can consider any data type elements as "leafs" of the data structure. Those "leafs" can be bound to the appropriate form fields.

Navigation tree node's "Generate Form" action creates a default form with child data type elements under the data structure element. The action can be invoked on the "UserInterface" root node or any other tree nodes. In case of the child tree nodes it uses the "Object" property type as current "model" type to build the form.

For example, "Generate Form" action on the testDs2 UserInterface will take only "attrDs2A" and "attrDs2B" elements. The action on the "dsArrayInstance" tree node will create fields for "attrA" and "attrB" – "leafs" for dsArray item "Element" type.

d Right click on "sample.ds.testDs1.UserInterface" and perform "Generate Form" for the testDs1 root.

Select the generated form as "Detail Form" for the UserInterface root node.

Note: leave "Detail UI" property empty. In this case the tree node's local forms will be used.

e Do the same actions for the "sample.ds.testDs2.UserInterface": generate a form and select it as "Detail Form" property.

3 "Refresh Node" action

Unlike orders, data structures do not have built-in notification mechanism when any of child items changes, for example, when a new instance of the element is added to the array or removed. We need to refresh the navigation tree and its nodes manually.

One of the ways to refresh the entire navigation tree or a particular tree node is to call “refreshNode” method on the root or that node. The “Refresh Node” action calls the method for the currently selected tree node.

a Create a User Action method “RefreshTreeNode” of the root of the “sample.ds.testDs2.UserInterface” navigation tree.

Note: Be careful naming the methods: do not override existing “updateNode”, “refreshNode”, etc. methods of the base metadata object. Overriding a base method without calling super method might break the tree node behaviour.

```
var selected = this.getSingleSelection();
// If nothing is selected, refresh the root (this)
if (selected == null)
  selected = this;
selected.refreshNode(true, true);
```

b Override “Menu” form of the “sample.ds.testDs2.UserInterface”;

Copy&replace “HorizontalLayout” object and add a new MenuItem to the layout.

Type menu item’s label and icon and select “RefreshTreeNode” as “Click method”.

4 Using notifications: “Add testDs1” action

The other way of refreshing nodes is using notifications. This way is preferable when you do not have access to the data structure UI or you have several UI controllers for the same data object.

The “Add” action for the “dsArray” is a good sample of using notifications. The action adds a new instance of the testDs1 data structure to the “dsArray” and notifies the corresponding tree node using “Notifier” API.

a Create a new User Action method “NewDs1” of the “sample.ds.testDs2.UserInterface” object:

```
// the model of the tree root is the testDs2 data
structure
var ds = this.model;
var count = ds.dsArray == null ? 0 : ds.dsArray.length;
// Init new testDs1 item in the dsArray by simply
accessing it
ds.dsArray[count].attrA = "Attribute A of testDs1 - " +
count;
ds.dsArray[count].attrB = "Attribute B of testDs1 - " +
count;
if (count == 0) {
  // Reload the entire root node: the “dsArray” container
  // might not have been added to the tree if accessing
for the first time
```

```
  Notifier.notify(ds, "reload");
}
else {
  // Notify that instance has been added. Parameters:
  // 1 - the parent data object; 2 - action type; 3 -
action child object
  Notifier.notify(ds.dsArray, "add", ds.dsArray[count]);
}
```

5 **How it works at runtime.**

When a tree node is created, it registers itself as a listener to the model data object (see “onModelAttached” method of the com.conceptwave.system.TreeNode):

Notifier.register(newModel, this, "onModelEvent");

Where parameters are:

a the source object of notification events;

b the listener;

c the name of listener’s method to invoke.

When we send a notification using particular data object (data structure in our case):

```
Notifier.notify(ds.dsArray, "add", ds.dsArray[count]);
```

the corresponding “dsArray” node of the navigation tree receives the notification event and the “onModelEvent” method of the node will be invoked.

The default implementation of the “onModelEvent” method processes some basic events: “update” to refresh the node, “add” and “delete” for children (then third parameter is the subject of the action), etc.

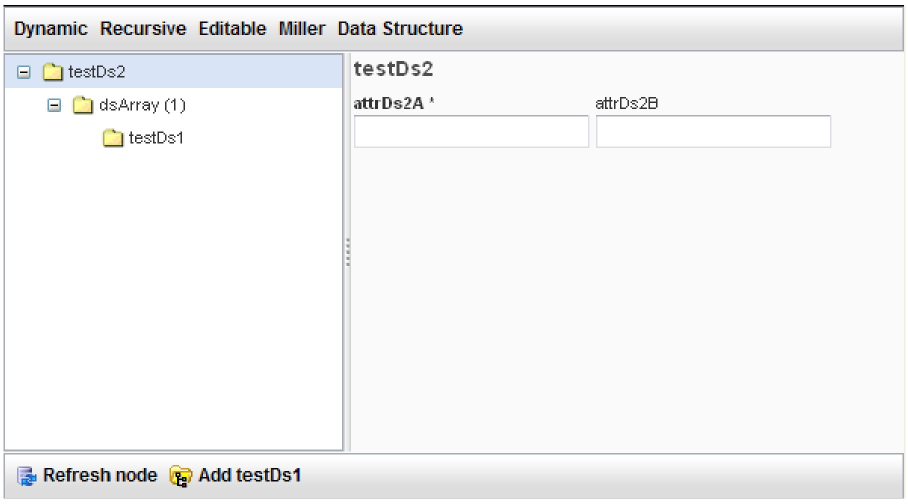
The “reload” action is our custom notification event type, so we have to process it:

d Override “OnModelEvent” method of the “sample.ds.testDs2.UserInterface”:

```
if (event.type == "reload")
this.refreshNode(true, true);
else
  this.cw$super_onModelEvent(event);
```

6 A new MenuItem to the “Menu” for of the “sample.ds.testDs2.UserInterface” root.

Type menu item’s label and icon and select “NewDs1” as “Click method”.



"Add testDs1" action at runtime

7 "Add dsDoc" action

The "Add dsDoc" action is similar to the "Add testDs1" action: it adds an instance of the "dsDoc" to the "docs" array.

a Create a new User Action method "newDocInstance" of the "dsArrayInstance" tree node of the "sample.ds.testDs2.UserInterface" navigation tree:

```
// The model of the “dsArrayInstance” tree node is a
“testDs1” data structure -
// an instance of “dsArray” element
var dsDoc = this.model;
// Add document to the "docs" array by accessing it
var count = dsDoc.docs == null ? 0 : dsDoc.docs.length;
dsDoc.docs[count].textField = "dsDoc " + count;
// Notify listeners
if (count == 0) {
  Notifier.notify(dsDoc, "reload", dsDoc.docs[count]);
}
else {
  Notifier.notify(dsDoc.docs, "add", dsDoc.docs[count]);
}
```

b Override the "onModelEvent" method of the "dsArrayInstance" node to process "reload" event type:

```
if (event.type == "reload")
  this.refreshNode(true, true);
else
  this.cw$super_onModelEvent(event);
```

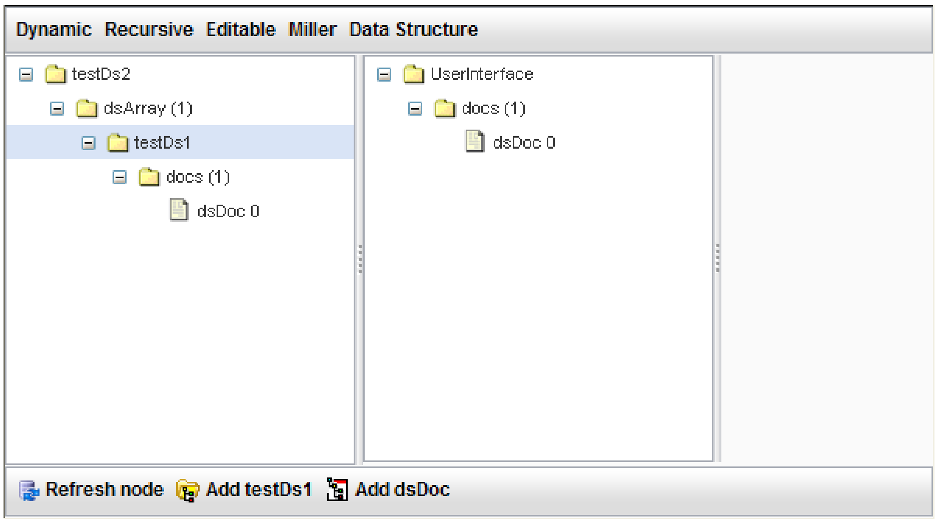
c Override the "Menu" form of the "dsArrayInstance" node, add a menu item "Add dsDoc" and select "newDocInstance" as "Click method".

d If you want to have the same action on the “docs” tree node level, follow the same steps for the “docs” tree node. In “newDocInstance” method of the “docs” node simply call the parent node method:

```
this.parentNode.newDocInstance();
```

e Go to the “sample.ds.testDs1.UserInterface” navigation tree and override the “onModelEvent” method to implement “reload” action for the “testDs1” UI as well:

```
if (event.type == "reload")
  this.refreshNode(true, true);
else
  this.cw$super_onModelEvent(event);
```



“Add dsDoc” action at runtime.

Note: The Detail UI for the selected “testDs1” tree node is set by default to “sample.ds.testDs1.UserInterface” (and Detail form: “Default”), that is why we see a testDs1 sub tree at the right side.

That sub tree is a completely different navigation tree object (sample.ds.testDs1.UserInterface) but it is bond to the same “testDs1” data structure instance as the “testDs1” tree node of the master tree.

When we sending notifications for the new “dsDoc” document, both trees receive the notifications and append “dsDocInstance” nodes.

f Updating “dsDoc” document visual key at runtime

“textField” leaf value of the “dsDoc” is a visual key of the document and also a label for the tree node.

Every time when we perform “save” action of a document, it automatically sends “update” notification, so the tree node label is updated automatically.

But the “dsDoc” document is not mapped to the DB and “Save” menu item is hidden. To update the visual key, send the notification manually (any time when field is changed in scripts or on “Update” menu action of the document UI):

```
Notifier.notify(document, "update");
```

10 Table element samples

10.1 Auto fit data [finder.AutoFitTable]

This sample shows how to make the table element auto fit its content and how height settings (Height and AutoFit Max Height) affect the behavior.

1. Create a new “finder.TestDoc” document; add 2 string variables: “name” and “description”; select “Generate key” checkbox and map the document to the database.

   Override “Default” form and add fields for “name” and “description” variables.

2. Create a new document finder: “finder.AutoFitTable” with empty “Input” property and the “Output” set to “finder.TestDoc” document.

   Go to the **View** tab of the finder and select the **Auto search** checkbox of the default view.

3. Create a new User Action method “GenerateDocs” of the finder UserInterface:
   ```
   // Generate 5 documents:
   var length = this.result.length;
   for(var i = 0; i < 5; i++) {
     var doc = new Document("finder:TestDoc");
     doc.name = "Doc " + (length + i);
     doc.description = "Description " + (length + i);
     doc.save();
   }
   this.searchAction();
   ```

4. Create a new User Action method “ClearTable” of the finder UserInterface:
   ```
   var length = this.result.length;
   for (var i = length - 1; i >= 0; i--) {
     this.result[i].deleteFromDB();
   }
   this.searchAction();
   ```

5. Override “Menu” form of the finder UserInterface and add menu items for the “GenerateDocs” and “ClearTable” actions.

6 Create “AutoFit” form under the finder’s UserInterface. Create same table as the table from the “Result” form. Set height of the table to some fixed width: 100px. This value will be the minimum table height. The value should be set to enable “Autofit Data” property.

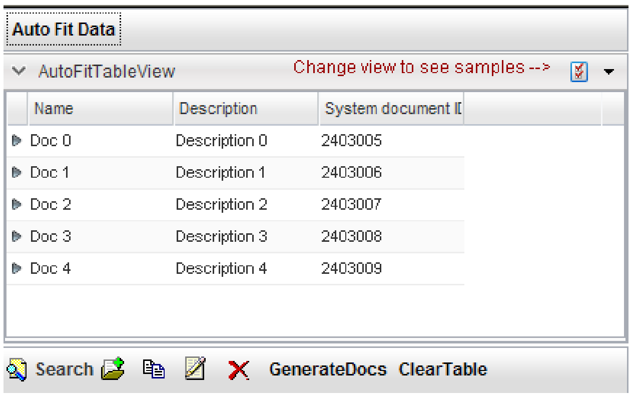
   Set “AutoFit Data” property of the table to “Vertical”.

7 Create “AutoFitWithMinHeight” form. Copy the table from the “AutoFit” form to the new form.

   Set “Auto Fit Max Height” property of the table element to 150px.

8 Create 2 extra views for the finder: “AutoFitView” with “AutoFit” result form and “AutoFitWithMinHeightView” for another form. Select “Auto search” checkbox for both views.

   When creating a new finder or when setting output document, the table element of the “Result” form is populated automatically. By default the table takes the entire space of the Result form:

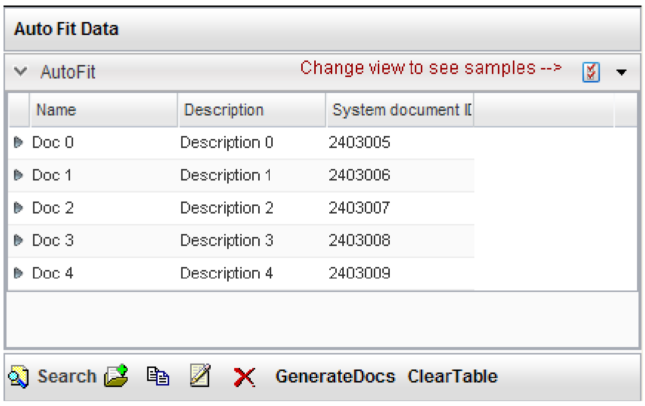

Auto Fit Data

AutoFitTableView   Change view to see samples -->

Name	Description	System document I
Doc 0	Description 0	2403005
Doc 1	Description 1	2403006
Doc 2	Description 2	2403007
Doc 3	Description 3	2403008
Doc 4	Description 4	2403009

Default height: 100%

Setting “Autofit Data” property of the table element, makes the table take the exact space for rows:

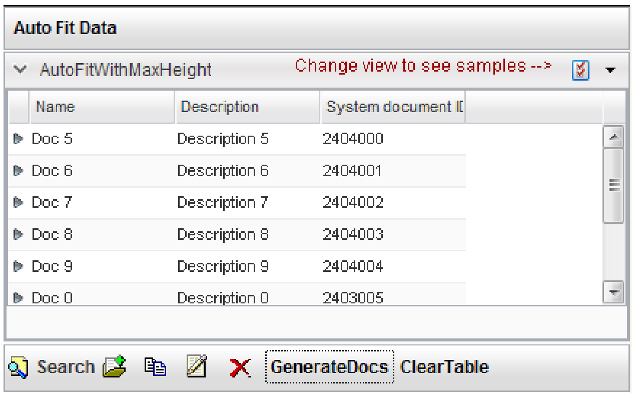

Auto Fit Data

AutoFit   Change view to see samples -->

Name	Description	System document I
Doc 0	Description 0	2403005
Doc 1	Description 1	2403006
Doc 2	Description 2	2403007
Doc 3	Description 3	2403008
Doc 4	Description 4	2403009

Vertical AutoFit

The “Height” property sets the minimum table height and the “Auto Fit Max Height” defines the maximum height that the table is allowed to take:


Auto Fit Data

AutoFitWithMaxHeight — Change view to see samples -->

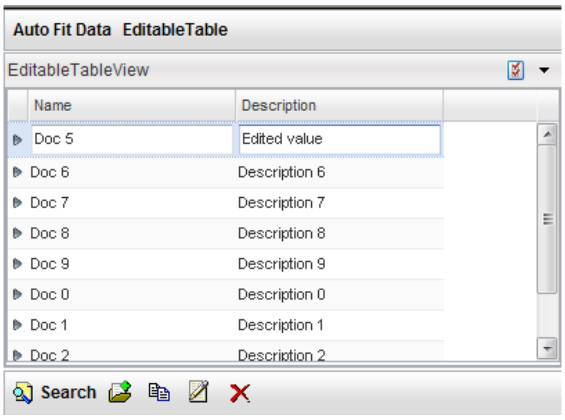
Name	Description	System document ID
Doc 5	Description 5	2404000
Doc 6	Description 6	2404001
Doc 7	Description 7	2404002
Doc 8	Description 8	2404003
Doc 9	Description 9	2404004
Doc 0	Description 0	2403005

Table set to Auto Fix Max Height

10.2 Editable Table [finder.EditableTable]

By default the finder table is read-only. To make it editable, set the “Editable” property of the form table element to “TRUE” or to some custom permission method.

The “finder.EditableTable” sample finder allows editing Name and Description variables in the table:


Auto Fit Data EditableTable

EditableTableView

Name	Description
Doc 5	Edited value
Doc 6	Description 6
Doc 7	Description 7
Doc 8	Description 8
Doc 9	Description 9
Doc 0	Description 0
Doc 1	Description 1
Doc 2	Description 2

1. Using “finder.TestDoc” document from the previous sample, create a new document finder “finder.EditableTable”. Leave the “Select input” property empty and set output to the “finder.TestDoc” document.

2. Go to the “Result” form of the finder and set “Editable” property of the table element to TRUE.

3. Remove cwDocId column.

4. Create a new user action “EditableTable” in the “app.app” application User Interface. Select the finder as User Action Object and Form -> Default. Create a new menu item on the application “Menu” form and select the “EditableTable” as click method.

10.2.1 Validation in Table

After the user completes editing the record in the table, the new values are being sent to the server and stored into the document variables (in memory). Then it validates the document and *only* if the document is valid, it runs document triggers and saved the document in the database.

If there are any validation errors (variable level), they will be displayed in the table in the corresponding column:

- To create a validation rule for the Description column go to the “finder.TestDoc”, Variables tab, select “description” variable and create a new validation method for it:
  ```
   document.description == 'a'
  ```
  and specify validation message: “Validation fired.”

10.2.2 Column Tiggers [finder.ColumnTriggers]

The following sample shows how to trigger checkbox in the table; capture image column click and display hover message for the table row.

10.2.3 Checkbox Column

1 Add a new Boolean variable “checkbox” in the “finder.TestDoc” document. We will use the same test document for the sample finder.

2 Create a new document Finder: “finder.ColumnTriggers”, specify “finder.TestDoc” as finder’s Output.

3 Create new User Action method under finder’s UserInterface “checkboxTrigger”:
```
if (this.result.selected != null &&
this.result.selected.length > 0) {
  // get the trigger document: selected row in table
  var document = this.result.selected[0];
  // Modify description variable
  document.description = "Checkbox value: " +
document.checkbox;
  document.save();

// force to refresh table to display modified description:
  this.result.updateList;
  Global.showUserMessage("checkboxTrigger fired.");
}
```

4 Edit the “Result” form of the finder: remove all created table columns by default.

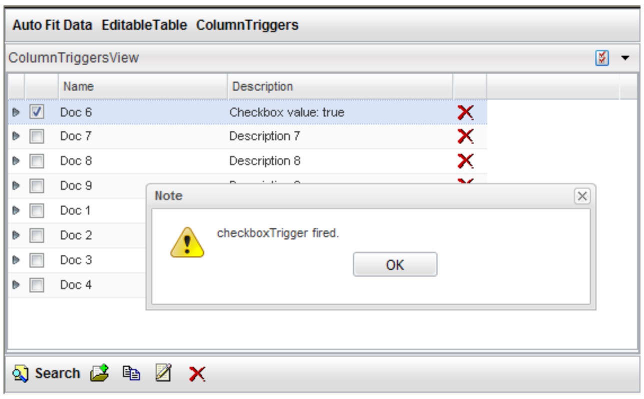
Set Editable property to TRUE.

5 Create a checkbox column and bind it to the “result.checkbox” Variable. Set the “Toggle Edit” property selected. Select “checkboxTrigger” method in “Run Trigger” property.

Note: if the property is not set, then it requires two clicks to change the checkbox value: first click to enter row edit mode and then click to actually change the checkbox value.

Set other properties: “Show Label”: false; “Width”: 30px.

6 Create two Label columns for “result.name” and “result.description” variables. Set desired width of the columns.


Auto Fit Data EditableTable ColumnTriggers

ColumnTriggersView

	Name	Description	
✓	Doc 6	Checkbox value: true	✕
	Doc 7	Description 7	✕
	Doc 8	Description 8	✕
	Doc 9		
	Doc 1		
	Doc 2		
	Doc 3		
	Doc 4		

Note

checkboxTrigger fired.

“checkboxTrigger” action

10.2.4 Image Button

The table does not need to be editable to invoke an action on image button click.

When invoking a trigger or an image click action, the action document is always the selected document in the table. To get the selected document use “this.result.selected[0]”.

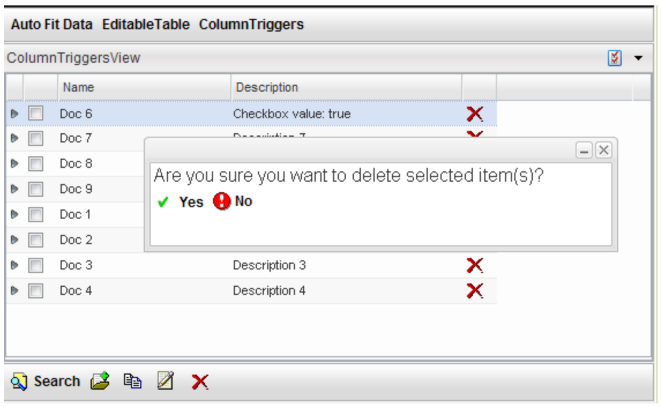
1 Create a new User Action method to fire an action on image column click **imageClick**:

```
// imageClick action: delete the selected document
this.deleteAction();
```

2 Add a new Image column to the table element of the “Result” form. Set properties:

“Image URL”: /cwf/MenuIconDelete.gif; “Click Method”: imageClick

“Show Label”: false; “Width”: 30px.



	Name	Description
	Doc 6	Checkbox value: true
	Doc 7	
	Doc 8	
	Doc 9	
	Doc 1	
	Doc 2	
	Doc 3	Description 3
	Doc 4	Description 4

Are you sure you want to delete selected item(s)?

✓ Yes ● No

Image click action

Dynamic Image

If you need to have a dynamic image, different for each row, you can bind the column Variable property to a script method and return the dynamic image path. The Script method must have return type: String and 1 parameter: “document” of table document type. The method is invoked for each row and the “document” parameter is the row document.

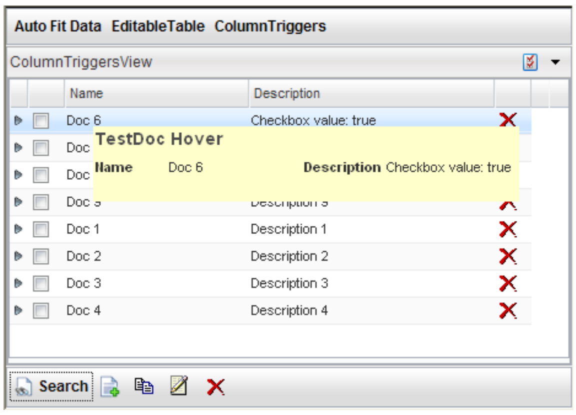
10.2.5 Hover Action

By configuring the hover properties of the table you can display a pop-up form as a result of a hover on a Finder result. There are four properties used to configure the hover option: Form, Method, Style and Variable.

The cwShowHover variable in the Finder User Interface is of type Boolean. It must be set to True when theHover Method is called. The value of this variable, once sent back to the browser, is set to null.

The following is a sample of the table hover in “finder.ColumnTriggers”. The hover displays the “HoverForm” form of the “finder.TestDoc” document.

Note: It is also possible to use a finder’s local form as a hover form.



	Name	Description
	Doc 6	Checkbox value: true
	Doc	
	Doc	
	Doc 9	Description 9
	Doc 1	Description 1
	Doc 2	Description 2
	Doc 3	Description 3
	Doc 4	Description 4

TestDoc Hover

Name Doc 6 **Description** Checkbox value: true

Hover action

1 Create a new “Hover Form” form under “finder.TestDoc” Forms folder. Populate the form with some form elements.

  Note: The hover form is not focusable; it disappears when mouse is out of row. It is better to use not-editable elements: labels, HTML content, URLs, etc. Technically you can put a text field there, but it will be not possible to edit the value in browser.

2 Go to the “finder.ColumnTriggers” finder’s UserInterface.

  Create a new “hoverDocumentUI” variable. Set type of the variable: “finder.TestDoc”.

3 Create a new method “onHover” (on the UserInterface level):

```
// onHover method
// this.result.hover is the hover row document
if (this.result.hover != null) {
  this.hoverDocumentUI =
        new finder.TestDoc.UserInterface(this.result.hover,
this);
  // set "cwShowHover" variable to true if need to show
hover form;
  // after the hover is shown the variable gets cleared.
  this.cwShowHover = true;
}
else
  this.hoverDocumentUI = null;
```

4 Go to the “Result” form and set hover properties of the table element:

  Hover Method: onHover;

  Hover Variable: hoverDocumentUI;

  HoverForm: hoverDocumentUI.Forms.HoverForm;

  Hover Style: CwCABody

11 Modifying URL mappings, style sheets and skins

There are several settings in Velocity Studio that enable you to modify the overall look and feel of the interface during design-time:

URL Mapping

Style Sheet

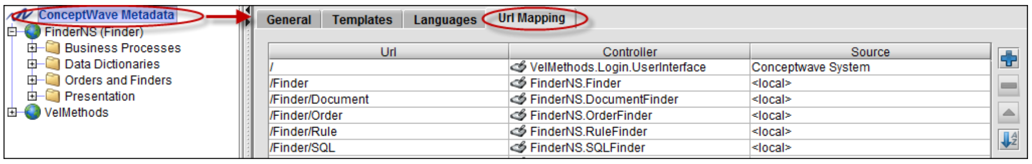
Skins

11.1 Configuring URL Mapping

The URL Mapping tab is found at the root node (Metadata) of your project. Setting up this information enables you to display User Interface objects (for example, Form, Page) to web clients, by assigning them to a relative URL path, that is relative to the following:

```
http://<hostname>:<port>/cwf/<XX>
```

where *XX* refers to the name of the Form or Page that you want displayed in the Web browser.



Url	Controller	Source
/	VelMethods.Login.UserInterface	Conceptwave System
/Finder	FinderNS.Finder	<local>
/Finder/Document	FinderNS.DocumentFinder	<local>
/Finder/Order	FinderNS.OrderFinder	<local>
/Finder/Rule	FinderNS.RuleFinder	<local>
/Finder/SQL	FinderNS.SQLFinder	<local>

For applications that have a URL mapping, the style sheet and skins properties are available at the Page object level of that application.

To setup the URL Mapping, do the following:

In the Navigation Tree, click the root node.

1 Click the **URL Mapping** tab.

2 Configure a mapping for the menu of your application.

3 In your application metadata, navigate to the **User Interface** node of the **Presentation** folder.

4 Navigate to the **Page** form.

5 Right-click and override the **Page** form. The **Page** properties open.

6 Set the **Stylesheet** property to the stylesheet that you want to display.

7 From the main menu, click **File** > **Save**.

11.2 Modifying the Style Sheet

The style sheet in Velocity Studio controls the appearance of the Page design of the runtime metadata. Upon opening Velocity Studio, the style sheet list box displays the default style sheet (cwf.css [/cwf/css/cwf.css]), which cannot be edited.

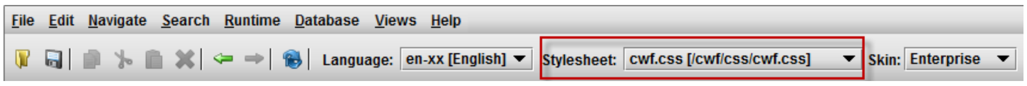
To customize the Page design, do the following:

1 In Windows Explorer, add a new CSS into the Resources folder of your project directory.

2 In Velocity Studio, open the project that contains the new CSS. Opening the project automatically loads all stylesheets.

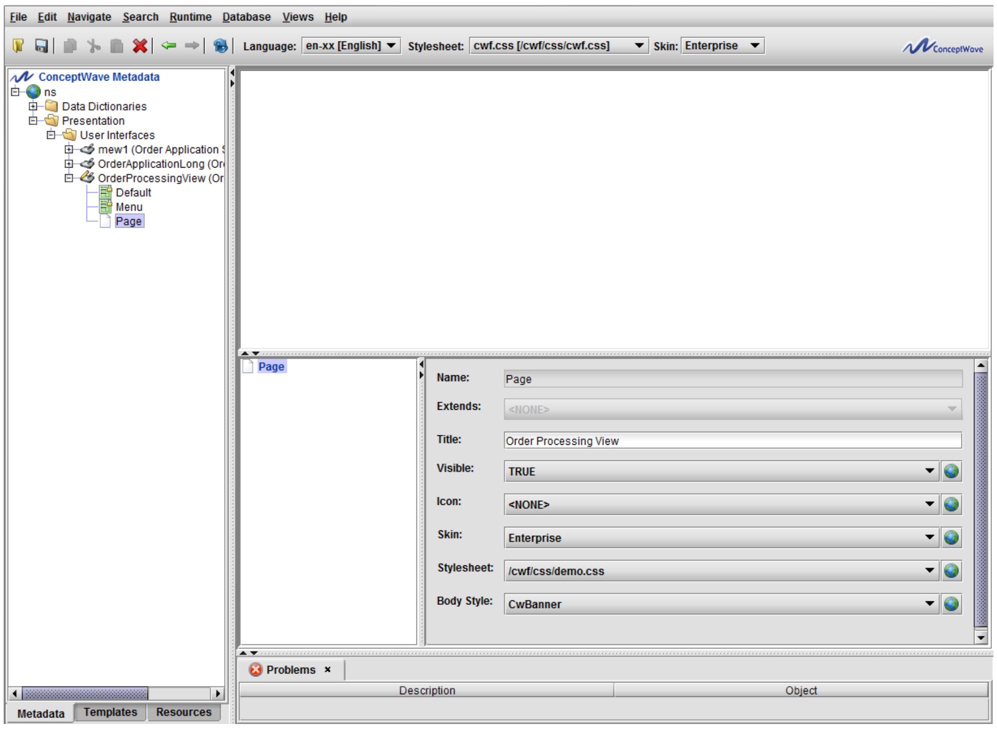
3 From the toolbar, click the Stylesheet list box and select your new CSS.

The style sheet is available from the toolbar and is used for rendering the preview pane. Changing a style sheet allows for look and feel changes at the form level. To modify the overall application's look and feel, you need to change the Skin.



Language: en-xx [English]
Stylesheet: cwf.css [/cwf/css/cwf.css]
Skin: Enterprise

If you have set the User Interface object using the **Extends** parameter `com.conceptwave.system.UserInterface`, the style sheet is available from the properties page of that Page. If you have set the User Interface object using the **Extends** parameter `com.conceptwave.system.application`, the style sheet is available from the properties page of that page providing that you have extended the page.



Language: en-xx [English]
Stylesheet: cwf.css [/cwf/css/cwf.css]
Skin: Enterprise

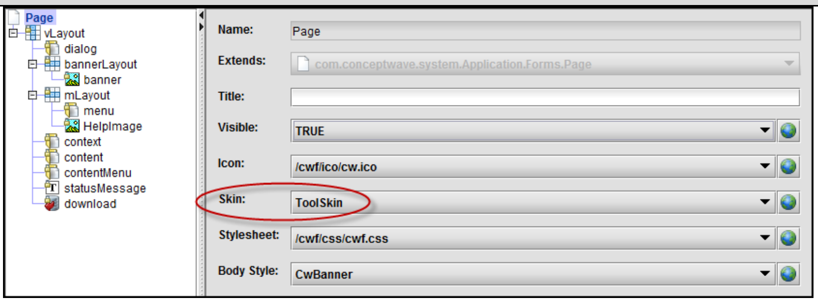
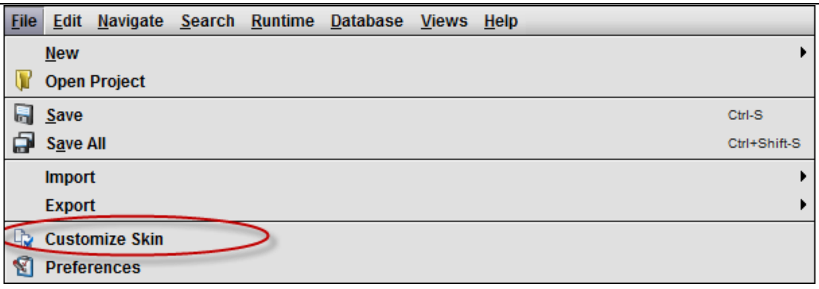
Name: Page
Extends: <NONE>
Title: Order Processing View
Visible: TRUE
Icon: <NONE>
Skin: Enterprise
Stylesheet: /cwf/css/demo.css
Body Style: CwBanner

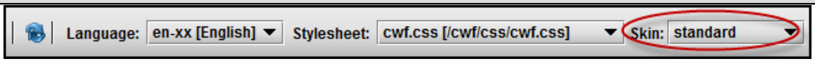
11.3 Skin

The default Skins directory is built into Velocity Studio. In general, the skins directory contains a style sheet and image files that controls the overall look and feel of the application. The Menu Bar provides a drop-down list of Skins that are available while configuring the application. Customizing the files in the Skins directory allow you to customize areas such as backgrounds, title bars, buttons, and other graphical interface elements. Skin customization allows for the overall control of the look and feel of the application including the ability to change *all* element styling including the images that are used to render the elements (for example, table or button). To customize the look and feel of a specific form, it is recommended to use a customized style sheet.

To customize a skin, select the Customize Skin option from the **File** Menu. This would create a new Skins directory in the metadata and copy all the skins files and folders, including the CSS which can be modified as necessary.

There are three different ways to access the Skins:

Skin type (accessed from)	Example
Page form	 Page vLayout dialog bannerLayout banner mLayout menu HelpImage context content contentMenu statusMessage download Name: Page Extends: com.conceptwave.system.Application.Forms.Page Title: Visible: TRUE Icon: /cwf/ico/cw.ico Skin: ToolSkin Stylesheet: /cwf/css/cwf.css Body Style: CwBanner From the Details pane, set the Skin for the Page to be viewed at runtime.
File menu Description	 File Edit Navigate Search Runtime Database Views Help New Open Project Save Ctrl-S Save All Ctrl+Shift-S Import Export Customize Skin Preferences The **Customized Skin** menu item allows you to copy an existing Skin and rename it to a new name in the application metadata under the skins/ folder (that is, `C:\<metadata home>\skins\`). The new Skin automatically saves and is available from the Page, toolbar or **File** menu. You can change the *js/image* files and then use this Skin for preview or runtime.

Skin type (accessed from)	Example
From the toolbar	 Language: en-xx [English] Stylesheet: cwf.css [/cwf/css/cwf.css] Skin: standard From the Toolbar, you can change the Stylesheet and Skin. However, this feature only provides a "preview" of the stylesheet and skin change. It does not actually set the Skin or Stylesheet for the form. To "permanently" set the Stylesheet and Skin for the application, the Skin and Stylesheet option is available at the Page level.

12 Acronyms

CSS – Cascading Style Sheet

13 Reference List

The following is a list of documentation for reference:

- *Velocity Studio User Guide*
- *Velocity Studio Configuration Guide*
- *Velocity Studio System Administration User Guide*
- *Catalog User Guide*

14 Trademarks

15 Disclaimer