NOKIA Developer

Archived:Developing your first WidSets widget

Archived: This article is archived because it is not considered relevant for third-party developers creating commercial solutions today. If you think this article is still relevant, let us know by adding the template {{ReviewForRemovalFromArchive|user=~~~~|*write your reason here*}}.

The article is believed to be still valid for the original topic scope.

This article gives step-by-step instructions on how to develop a widget for WidSets. The example application used in this article prompts the user to select a country flag and then displays a greeting accordingly.

Create a widget project

Widget development does not need a project file. Instead, the component files of each widget are simply stored in a folder which can be freely named. All required files can be found inside the Flag Selector widget ZIP-file File:Flagselector.zip. The Flag Selector widget folder structure and component files are as follows:
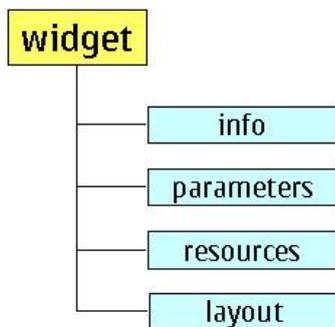
```
\flagselector\
  widget.xml
  flagselector.he
  web_icon.png
  web_minimized.png
  web_miximized.png
  background.png
  finland.png
  eu.png
  un.png
```

Create the widget.xml

The example widget.xml file contains different types of widget information. To keep things simple, the file contents have been split into a number of subareas and only essential areas are discussed in detail. For more information about the widget.xml structure and contents, refer to other articles in.

First, the format of a widget.xml file is explained, after which each element block is discussed. Element block means the block of information within the mentioned element's start tag and the end tag according to XML rules.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<widget spec_version="2.1">
  <info>
    <name>Flag Selector</name>
    ...
  </info>
  <parameters>
    ...
  </parameters>
  <resources>
    <img scale="true" src="finland.png"/>
    <img scale="true" src="eu.png"/>
    <img scale="true" src="un.png"/>
    <img src="bkg.png"/>
    <code src="flagselector.he"/>
    <stylesheet>
    ...
    </stylesheet>
  </resources>
  <layout>
  </layout>
</widget>
```

The <root> element

The widget root element is mandatory in the widget.xml content of every widget. It has one attribute and it must declare the WidSets specification version. Currently WidSets supports the specification version 2.1. Widget configuration is backwards compatible. Changes are announced in the WidSets developer forum.

More specific information about the widget specification can be found in the WidSets wiki category. Also, the widget.xml file MUST be encoded with UTF-8 encoding. Thus, the Flag Selector widget would have the following XML code lines in its widget.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<widget spec_version="2.0">
  ...
</widget>
```

The <info> element

It is good practice to start by editing the widget's metadata, such as the name of the widget, its version, the author's name, and a short description of the widget. These properties are defined in the info element block. The element block has a number of mandatory and optional sub-elements. This article discusses only the mandatory fields.

- The widget's name is used as a name tag to identify the widget within WidSets.
- Version numbering is a user-defined value to set the current widget version under development.
- The author name does not have to be the name of the uploader and/or publisher of the widget.
- Short and long descriptions are used in free text search.

```
<info>
  <name>Flag Selector</name>
  <version>1.0</version>
  <author>Developer</author>
  <shortdescription>Displaying some flags</shortdescription>
</info>
```

The <parameters> element

Parameters can be used in numerous different use cases. Basically, parameters are a set of helpful dynamic containers for settings-related data. Parameter values are kept on the WidSets server, and, for example when reloading a widget, it always defaults to these parameters.

Values that are kept in the client-side storage are always scrapped when the widget is reloaded. Usable parameters need to be defined in the widget.xml and can then be used and changed in the code of the widget.

Widget parameters are also tightly integrated into the WidSets client UI and parameter values can be changed on each widget's Settings page (Widget/Settings). Additionally, developers have control over the editability, visibility and, of course, name and type of parameters while creating their own widgets. Parameters do not necessarily have to be sent to the mobile device, either.

```
<parameters>
  <parameter type="string"
             name="widgetname"
             description="Name of widget"
             help="This is the name of the widget"
             editable="false"
             sendtomobile="true"
```

```
              visible="true">
    <value>
      Flag Selector
    </value>
  </parameter>
</parameters>
```

The <resources> element

A widget may have some graphics and styles to be displayed in its user interface components. It can also contain WSL code to perform some interactivities.

The resources element block is where you can define all these resources for your widget.

To define the graphic resources, use the <img> element and define its attributes as follows:

```
<img scale="true" src="someimage.png"/>
```

The scale attribute is set to "true" or "false" to tell the WidSets server whether or not to rescale the image size to be suitable for a mobile device's screen resolution while the widget is downloaded to that mobile device.

Note that the CSS structure used in WidSets is compliant with W3C CSS specifications. However, WidSets uses its own version of CSS properties and values for defining widgets' UI styles. All style definitions are defined within the stylesheet element. The style definition syntax is made up of three parts: a style name, a property, and a value or values (note that some properties can have multiple values).

The following example styles are defined and used throughout the Flag Selector widget File:Flagselector.zip:

```
<stylesheet>
  bkg {
    background: vgradient white silver;
  }

  frame {
    background: solid silver;
  }

  flag {
    align: hcenter vcenter;
    border: 2 2 2 2;
    border-type: rectangle red;
  }

  nametext {
    align: hcenter vcenter;
    color-1: green;
    font-1: medium bold;
    background: vgradient silver white;
  }

  greetingtext {
    align: hcenter vcenter;
    color-1: blue;
    font-1: medium bold;
    background: vgradient white green;
  }

  about {
    align: hcenter vcenter;
    color-1: blue;
    font-1: large bold;
    background: solid orange;
  }
</stylesheet>
```

CSS properties and values are quite self-explanatory. For example, in the code snippets above, the style named bkg has been defined for the background of the widget. The bkg style has the background property assigned with a value of vertical gradient [vgradient] with the color starting from white and gradually changing to silver. For more information about styles and their usage, refer to the documentation in Archived:Stylesheet in WidSets.

If a widget has extension functions, they should be implemented in a WSL file with the filename extension .he (for example, flagselector.he), and the source file must be declared in the code element as shown below:

```
<code src="flagselector.he"/>
```

Note: Each widget can only have one scripting source file.

The <layout> element

Each widget is constructed of views and the layout element embraces all views of a widget, which has two possible direct child element types as the view and/or webview element.

The layout element also has a mandatory attribute, "minimizedheight", which is used to define the height of the widget when displayed in the WidSets client view (the WidSets dashboard).

Each view or webview element has attributes (class, id, top, right, bottom, left, width, height) and possible children elements (label, text, decorate, img, script), as shown below. The Flag Selector layout definition is as follows:

```
<layout minimizedheight="60px">
  <view id="miniView">
    <img src="minimized.png"/>
  </view>
```

```
<view class="bkg" id="mainView">
  <script id="flag" top="10%" right="80%" bottom="60%" left="20%"/>
  <script id="name" top="65%" right="100%" bottom="80%" left="0%"/>
  <script id="greeting" top="80%" right="100%" bottom="100%" left="0%"/>
</view>
</layout>
```

The code above defines the minimum height of 60 pixels for the widget to be displayed on the dashboard. The minimized view of the widget is represented by the minimized image, where the resource is minimized.png.

Another view of the widget is called the main view, which is identified by the id attribute (that is, id="mainView"). It is designed to have the main view's background style as defined in the bkg stylesheet (that is, class="bkg"). The mainView element has three children and in this case, all are script elements. Each script element also has an id attribute for identification, and other attributes for defining its position on the screen. These UI components are not given a style yet, since the style is set when constructing the UI components in the WidSets Scripting Language (WSL) script file as explained in the next section.

Create and implement the WidSets Scripting Language file

This section explains how to create a WSL file and implement some basic WidSets functions to the Flag Selector widget.

Create a new text file and define the only class of the widget as follows:

```
class
{
}
```

Save this file as flagselector.he. It is ready to define and implement any functions within the class above.

Start with the startWidget() function

```
void startWidget()
{
  setMinimizedView(createView("miniView", null));
}
```

Within this function, call the createView() function and pass two parameters as required, the identifier of the view and the context (in this example, the context is set null). The createView() function returns an object of Flow type, which is input for the setMinimizedView() function. As a result, you will see the minimized image of the widget on the dashboard.

Continue with the openWidget() function,

```
Shell openWidget()
{
  flags = new List()
    .add(getImage("finland.png"))
    .add(getImage("eu.png"))
    .add(getImage("un.png"));

  mainShell = new Shell(createView("mainView", null));
  updateScreen();
  return mainShell;
}
```

There is a global variable called 'flags' and it is an instance of the List object. Creating the list and adding images to that list is convenient if you want to display the image later. Note that this is only a demo to display some images as the widget's graphic resources. It is not the technique to implement a widget that displays pictures as content downloaded from a server, such as Flickr.

There is also a global variable named 'mainShell' which is an instance of the Shell object. This time call the createView() and pass the "mainView" as a parameter to create the main view of the widget. updateScreen() is a proprietary function that is implemented later to update the data and display it on the screen. The openWidget() function then returns the "mainShell" to the system.

Calling createView() triggers the system to call the createElement() callback function. The implementation of the createElement() function in the widget is as follows:

```
Component createElement(String viewId,
                        String elementId,
                        Style style,
                        Object context)
{
  if (elementId.equals("flag")) {
    flag_flow = new Flow(getStyle("frame"));
    flag_flow.setPreferredSize(-100, -100);
    return flag_flow;

  } else if (elementId.equals("name")) {
    name_flow = new Flow(getStyle("nametext"));
    name_flow.setPreferredSize(-100, -100);
    nameStr = new Text(getStyle("name"), "");
    name_flow.add(nameStr);
    return name_flow;

  } else if (elementId.equals("greeting")) {
    greeting_flow = new Flow(getStyle("greetingtext"));
    greeting_flow.setPreferredSize(-100, -100);
    greetingStr = new Text(getStyle("greetingtext"), "");
    greeting_flow.add(greetingStr);
    return greeting_flow;

  } else {
    return null;
  }
}
```

Since only one view has been defined in the layout element block in the widget.xml file, (that is, the "mainView" - "miniView" can be ignored because it does not have other UI components than the minimized image) you do not need to detect the viewId. Use "if" and "else if" statements to check the element's id, in this case, the script element's id, which was defined in the widget.xml within the "mainView" element.

If you detect the element's id, which is equal to "greeting",

```
else if (elementId.equals("greeting"))
```

Construct the greeting UI component as follows:

There are two global variables, "greeting_flow" and "greetingStr, as instances of the Flow object and of the Text object respectively.

The "greeting_flow" is a container which holds the "greetingStr" when the add() function is called and passed "greetingStr" through the function's parameter.

```
greeting_flow.add(greetingStr);
```

The getStyle() function returns a Style object named "greetingtext". The style was defined in the stylesheet in the widget.xml file. When creating the container, give it the style that returned from the getStyle() function (that is why there was no need to specify the class attribute while defining the "greeting" script element). The style "greetingtext" is also set for the "greetingStr" instance.

```
greeting_flow = new Flow(getStyle("greetingtext"));
greetingStr = new Text(getStyle("greetingtext"), "");
```

The createElement() function then returns the greeting_flow to the system. The createElement() function is called as many times as there are child elements of the view element being created.

The next steps are to create a menu for the widget. Start just with a simple menu with a "Back" option associated with the right softkey of a mobile device. To do this, define the widget's menu properties in the global scope as follows:

```
const int CMD_BACK = 1;
MenuItem BACK = new MenuItem(CMD_BACK, "Back");
```

Then implement two functions to handle menu events as follows:

```
MenuItem getSoftKey(Shell shell, Component focused, int key)
{
  if (key == SOFTKEY_BACK) {
    return BACK;
  }
  return null;
}
void actionPerformed(Shell shell, Component source, int action)
{
  switch(action)
  {
    case CMD_BACK:
      popShell(shell);
      break;
  }
}
```

From the code above, you can see that there is a menu command id CMD_BACK and a menu item BACK, which is assigned with the CMD_BACK command id and the text string "Back" as its name (this name will appear on the right side of the softkey pane of the widget).

The getSoftKey() function is called every time a user presses one of the two softkeys of a mobile device. Since the example widget is a simple one with only one view, the "shell" and "focused" parameters passed along in this function can be ignored. Now detect which one of the two softkeys was just pressed.

WidSets provides two identifiers, SOFTKEY_BACK and SOFTKEY_OK for the right softkey and for the left softkey, respectively. In the example widget, the right softkey is associated with the BACK menu item. That is why the BACK menu item is returned to the system when the "key" equals to SOFTKEY_BACK. At this point, if the user clicks on the left softkey, "null" is returned and nothing happens.

After the getSoftkey() function is called, the system calls the callback actionPerformed() function so that you can detect which command action the user has just selected and perform appropriate actions. If the user presses the right softkey, CMD_BACK is the command action, and the widget returns to the minimized mode. To do so, the popShell() function is called to pop the current view (contained in the "shell" parameter) out of the view stack. Since the widget currently has only one view on the stack, it returns to the minimized mode.

When a shell is created in the openWidget() function, it is returned to the system and automatically pushed into the widget's view stack. Menu and view stack manipulation are discussed later when more advanced features are added to the widget.

The following functions handle the image and texts to be displayed on the screen.

```
void updateScreen()
{
  if (index == 3) {
    index = 0;
  }

  switch (index) {
    case 0:
      nameStr.setText(FINNISH_TEXT);
      greetingStr.setText("Terve Suomi!");
      break;
    case 1:
      nameStr.setText(EU_TEXT);
```

```
            greetingStr.setText("Hello European Union!");
            break;
        case 2:
            nameStr.setText(UN_TEXT);
            greetingStr.setText("Hello United Nations!");
            break;
    }
    changeFlag(index);
    index++;
    flushScreen(true);
}

void changeFlag(int index)
{
    flag_flow.remove(0);
    Picture flagImage = new Picture(getStyle("flag"),
                                    Image(flags[index]));
    flagImage.setFlags(VISIBLE);
    flagImage.setPreferredSize(-100, -100);
    flag_flow.add(flagImage);
}
```

In the updateScreen() function, detect the index value and set the texts to be displayed accordingly. The index value is increased by 1 every time this function is called and it is reset to 0 when it rises to 3. The changeFlag() function and the index value is passed via the function parameter.

In the changeFlag() function, the first item in the "flag_flow" container is removed (this is actually not necessary for the first time the function is called since the container is empty when it is constructed in the createElement() function). A "flagImage" variable is declared and created as an instance of the Picture object. The reference image source is fetched from the "flags" list at the position defined by the index value. The image of a flag is loaded by casting the image source to an Image object.

```
    Image(flags[index]);
```

"flagImage" must also be set to be visible and its preferred size. Finally, add the "flagImage" to the "flag_flow" container and return to the updateScreen() function, where the flushScreen() function is called to force the system to repaint the widget's view immediately.

Create the widget graphic resources

The sizes of each graphic resource are discussed in Archived:Widget files. Besides the resource files for WidSets Web site and the minimized image, the example widget has three other graphic resources: the Finnish flag, the flag of the European Union, and the flag of the United Nations.

Prepare the widget for uploading to WidSets server

If you want to deploy a widget to the WidSets server by using the upload form in the WidSets Web site, the widget project files must be compressed in a ZIP file before it can be uploaded. The ZIP file must not contain the project folder itself. To achieve this, select all the project files and zip them rather than click on the project folder and zip it.

Another (easier) way to upload a widget to the WidSets server is to use the SDK. All you need to do is to login to the server, then at the prompt, issue the "run" command as follows:

```
    devkit run flagselector
```

Where flagselector is the Flag Selector widget project's folder.

The SDK first zips the files in that directory, uploads them to the server, and then starts the emulator. When the emulator is started, it automatically re-uploads the widget whenever you change the widget.xml file, and removes the possible old widget with the same name from your dashboard. Also if you change your WSL file the while emulator is running, the widget is reloaded with a new script.

Load the widget to a target device

If you have installed the WidSets client into your mobile device, start the client. When you have logged on the WidSets server, the widget is automatically uploaded to the WidSets dashboard. If the WidSets client is already started and you uploaded it using the Web form, you need to click on the "synchronize" button on the WidSets manager to load it to your device.

Note: If you are using the same widsets.com user account on both devkit and your mobile device, you need to close the emulator or switch to offline mode before using widsets.com on the mobile device, and vice versa. Only one device can be connected to the server at the same time using the same user account.

Publish the widget

Uploading your widget to the WidSets server only makes your widget available in your own account. If you want to publish your widget in the WidSets library, use the WidSets manager with following steps:

1. Select the widget from the manager board. The Option button below the board is enabled automatically.
2. Click on the Option button. This brings you to the next page where you can choose to publish your widget as a new widget or to update if it was published before.
3. Select "New widget" and click OK. This again brings you to the next page, where you should fill in the description and other information, and accept the disclaimer before WidSets can actually allow you to publish your widget.

Add some advanced functions to the widget

Now you can add some advanced functionality to the example widget.

Handling menu options

First, add a menu option with several menu items associated with the left softkey. To achieve this, some more menu properties must be defined in the global scope as follows:

```
   const int CMD_MANUAL = 10;
   const int CMD_TIMER = 11;
   const int CMD_ABOUT = 12;

   MenuItem OPTIONS = new MenuItem(OPEN_MENU, "Options");

   Menu MENU = new Menu()
     .add(CMD_MANUAL, "Manual")
     .add(CMD_TIMER, "Timer")
     .add(CMD_ABOUT, "About");
```

Now the widget has three more menu command ids defined and a menu item OPTIONS. It is associated with a system-defined constant OPEN_MENU, which tells the system that this is an open menu. The OPTIONS menu item also has a name "Options", which is shown on the left side of the softkey pane.

You also need to create a menu called MENU and add the three command ids as shown in the codes above. Each item in the menu has a command id and a name.

Next, modify the two functions implemented in section "Create and implement a WSL file" and implement one more system function as follows:

```
   MenuItem getSoftKey(Shell shell, Component focused, int key)
   {
     if (key == SOFTKEY_OK) {
       return OPTIONS;
     }
     else if (key == SOFTKEY_BACK) {
       return BACK;
     }
     return null;
   }

   void actionPerformed(Shell shell, Component source, int action)
   {
     switch(action) {

       case CMD_BACK:
         if (scheduler != null) {
           scheduler.cancel();
         }
         popShell(shell);
         break;

       case CMD_TIMER:
         scheduler = schedule(2000, 2000);
         break;

       case CMD_MANUAL:
         if (scheduler != null) {
           scheduler.cancel();
         }
         updateScreen();
         break;

       case CMD_ABOUT:
         MENU.enable(CMD_TIMER, false)
             .enable(CMD_MANUAL, false)
             .enable(CMD_ABOUT, false);
             ...
         break;
     }
   }

   Menu getMenu(Shell shell, Component source)
   {
     return MENU;
   }
```

In the first function, when the left softkey was pressed (that is, the key that equals to SOFTKEY_OK), the OPTIONS menu item is returned to the system. Since this is an open menu (as explained previously), the system calls the getMenu() function to get a menu that the widget wants to open. In this case, the MENU, which was created previously, is returned. As a consequence of the action above, the widget has its Options menu opened with three selections named "Manual", "Timer", and "About" as defined for the widget's menu.

Detecting user's actions within the switch case loop in the actionPerformed() function goes as explained in section "Create and implement a WSL file". Handling the user's actions and enabling/disabling the menu commands is discussed in the following subsections.

Using a timer

If the user has just clicked the "Options" menu and selected the "Timer" command, the user's action is detected and a timer using the schedule() function is created. The example widget uses this timer to change the displayed flag and texts every 2 seconds.

To do so, set the timer and implement the callback function as follows:

```
   Timer scheduler;
   ...
   case CMD_TIMER:
     scheduler = schedule(2000, 2000);
     ...

   void timerEvent(Timer timer) {
     updateScreen();
   }
```

The schedule() function used in this example takes two parameters (there are several overloaded schedule() functions). The first parameter is the delay time after which the timer expires. The second parameter is the time interval, after which the timer repeatedly strikes. When the timer strikes, the system calls the callback timerEvent() function, where updateScreen() is called to change the displayed flag and texts.

The timer is a "system-expensive" resource, and so you should carefully design your widget to limit the usage of such an expensive resource and/or free the resource when you no longer need it. You will see throughout the example that the scheduler is cancelled when switching to another view, or when the widget is in its minimized mode.

Creating a new view dynamically

The widget's view can be created dynamically without the need of defining a view element in the widget.xml. Next, create the "About" view dynamically for the Flag Selector widget, continuing from where the user has just clicked the "Options" menu and seleted the "About" command.

```
case CMD_ABOUT:
{
  MENU.enable(CMD_TIMER, false)
      .enable(CMD_MANUAL, false)
      .enable(CMD_ABOUT, false);

  if (scheduler != null)
      scheduler.cancel();

  Text about_text = null;
  Flow about_flow = null;
  Shell infoShell = null;

  about_flow = new Flow(getStyle("About"));
  about_flow.setPreferredSize(-100, -100);

  about_text = new Text(getStyle("About"), "Congratulation!");
  about_text.setFlags(VISIBLE|WRAP);
  about_text.setPreferredSize(-100, -100);
  about_flow.add(about_text);

  infoShell = new Shell(about_flow);
  pushShell(infoShell);
  break;
}
```

Because the "Options" menu is not needed in the "About" view, all the menu commands of the "Options" menu are hidden by calling the enable() function of the Menu object and passing the menu command id and the flag set to "false". This is done for the three menu commands as shown in the code above.

The timer is not needed anymore when entering the "About" view. That is why it is checked if the timer is active, and canceled to release the resource.

The next step is to create the "About# view and display it. Define three local variables as an instance of Text, Flow, and Shell objects. You also need to initialize them with "null# value (WSL strict rule).

The construction of the "about_flow" container and the "about_text" object is done the same way as explained in the createElement() function implementation. What is new here is the "infoShell" which contains the "about_flow" and pushed into the view stack via the pushShell() function.

When a new view is pushed into the view stack, the system automatically updates the widget's view with the new view. When the user clicks the "Back" menu, we return to the main view simply by calling the popShell() function:

```
case CMD_BACK:
  if (scheduler != null) {
      scheduler.cancel();
  }
  popShell(shell);
  break;
```

If the "Back" menu is clicked in the main view, there are no more views in the view stack, and the widget returns to the minimized mode on the dashboard to wait for user's action to open it again.

You have now successfully developed your first complete WidSets widget. Congratulations!