Creating Skins for MP2

Table of Contents

Overview

This section introduces into the development of skins for MediaPortal 2. It will focus on the skinner's view. To understand the reasons for some of the skinning precepts given here, and to get some technical background, you should read the Skin Engine page.
When reading, you should always check the explained concepts against the currently available skin code (of the default skin, for example).

Details

Skins and themes

When we talk about *skins* in general, we think of two things:

- *Screen files* describing what is shown in the corresponding screen (for example the screen file for the home screen shows the header title, the date and time and the menu at the left side)
- Associated *themes* containing the descriptions of control styles, colors, graphics and other items which are exchangeable (for example the default theme defines styles for controls like the buttons, list views, check boxes and others)

Technically, it is not necessary to separate the theme from the skin (style resources could also be declared inside the skin files, for example), but a full-featured skin typically is devided into those two parts to make the theme easily exchangeable.
You can create a completely new skin with one or more complete themes, or you can inherit several items from another skin. You even can enrich an existing skin with new themes. So the term *skinning* refers to the job of creating or deriving one or more themes and/or creating screen files.

Skins and executable code

Some areas of screens contain static content, but most areas show dynamic content which comes from the system. There are label contents (e.g. the time and date in the upper right corner of the default skin), lists (the menu or the media navigation), tree views and other controls which show data provided by so called *models*. The technique to bind the skin's controls to data properties from underlying code is called ***data binding***.
Concerning skinning, we can devide the system in three parts: The skin, its view models and the core system.
The core system mostly provides the data which should be presented by the skin, but neither its data format is really nice to handle, nor is the skin able to bind to it properly.
At that point, models (or: view models) come into play. View models are executable code which have been implemented by a plugin developer. They preprocess all kinds of data from the system and make it available for the skin to data-bind to it.
The skin then can use special data binding expressions which tell the SkinEngine to create references from the skin to the underlaying model data (and vice-versa, for update notifications).

Hint: Perhaps you know the MVC (model-view-controller) pattern. Don't mix up the MediaPortal 2 approach with an MVC approach. They are similar but different. In the terms of MVC, our 'models' have a similar job as the MVC 'controller'. Our base system and all other data sources together can be compared with the MVC 'model'. The best is, you forget about MVC for the time you're writing plugins for MP2.

Another important thing to know is, models and skin parts will NEVER be part of the core application. They are always added to the system by a plugin. The MediaPortal 2 core system is completely free of hard-coded skin components.

Skins and plugins

A skin consists of multiple logical parts. Each plugin brings its own (logical) screens with it. When skinning, you should be aware that EACH visible part of MediaPortal 2 is backed by at least one plugin. Or in other words: Without plugins, you cannot see anything from MediaPortal 2.
That means when creating a new skin, which should provide several new screen files, you must have multiple plugins in mind, whose logical screens should be presented with the new new skin. Optimally, a good skin would cover the screens of EACH plugin, but it is obvious that a skinner cannot create screen files for all possible plugins. We had that problem in mind when creating our default skin; it allows skinners to extend it without the need to override every available screen from each available plugin. That is possible because the frame of each screen file in the default skin is drawn by a master template file, which is included by each screen. Only the *client* area layout is described in the screen files. At runtime, the SkinEngine puts both parts together by loading the screen file and merging it into the master template file to create the final screen.
So, using the master template approach, in each screen, the frame controls look the same and the look and feel can be unified all over the application.
Hint: The easiest way to create a different looking skin is to just override the master template files.

Plugins and the default skin

Unlike in MediaPortal 1, in MediaPortal 2 the files of the default skin are spread over all plugins. If you are looking for the skin files of the default skin in the installed MediaPortal 2 application, you don't find it in one single directory. Instead, you have to look into each plugin which provides its own logical screens (and thus must provide its own skin contents). Each plugin is forced to at least contribute ITS screen files to the *default* skin.
Why is it that complicated? The answer is, MediaPortal 2 is designed to achieve its flexibility by being open for all kinds of plugins. End users will typically have installed many extension plugins. Each plugin folder is self-contained, i.e. it is sufficient to place a plugin folder into the *Plugins* directory and the system will use it.
The separation of the skin into all of its plugins also helps to keep an overview about dependencies. If a skin file is dependent on some style or control which is implemented by another skin, it is simple to see that dependency because the style is defined in a file which is located inside the other plugin.
So keep in mind: When creating a plugin, you MUST also implement the default screen files for it.
*Hint: The integration of a new skin part into the rest of the system is done by declaring **workflow states** and **menu actions**, which will be explained later.*

(Logical) screens

When plugin code and a skin are loosely coupled, you need a fixed interface between them. The "communication unit" with the skin in MediaPortal 2 is a *screen*.
If the system wants to present a special state to the user, it switches to its screen. A screen is the graphical representation of a special UI navigation state. When a plugin developer designs the UI for his plugin, he designs a *workflow*. That workflow consists of several internal workflow states and their graphical representations, the screens.
Examples for screens are the *home* screen, the *LocalMediaNavigation* screen and the *ShowHomeServer* screen.
Strictly speaking, the term *screen* is used for two different things:

- A concrete screen implementation aka *screen file* or *screen implementation*, for example the file *home.xaml* of the default skin
- A logical screen, i.e. the name and the job of a unit which has a concrete implementation in at least the default skin. For example *home* is a logical screen name. Another one is *LocalMediaNavigation*. So, a logical screen is uniquely defined by its name and it has a special job. For example the *home* screen has the job to welcome the user.

Depending on the context, the term refers to a logical screen or a concrete screen implementation.

Skins and the workflow engine

In earlier versions of MediaPortal, transitions from one screen to another were made by all possible components: Sometimes the screen implementation loaded another screen by providing a call directly to the SkinEngine, sometimes the underlaying model switched the screen, sometimes the SkinEngine switched screens and so on.
In MediaPortal 2, it is more regulated. We introduced the concept of *workflow states*. A workflow state is one defined state of the UI application part. Examples for workflow states are the *Home* state, the basic navigation state of the music library, each navigation step during the media navigation and so on. You can see workflow states at the navigation bar in the default skin. Each of the arrows represents an active workflow state. Workflow states are stacked upon each other, so when you navigate from the home state to the media navigation, you see both states in the navbar.

Workflow states mostly correspond 1:1 to screens, i.e. the *Home* workflow state has exactly one screen, the *home* screen. But other workflow states might switch its screens depending on some other application state. For example the same workflow state could show its 'normal' screen or a slightly modified screen, or a even dialogs on top of it. Normally, each workflow state simply has a screen hard coded in its workflow state definition.

The UI navigation is mostly controlled by the workflow engine. The main menu, which you can see at the left side of each screen in the default skin, contains so called *menu actions* which can be of different type. One type of menu action is the *PushNavigationTransition*, which means it will push a workflow navigation state onto the workflow navigation stack. The *Music* action is such a menu action, for example.

There are several workflow navigation actions which can be used in the main menu. There are also other system components which can trigger a workflow navigation. Typically, screen changes are triggered by a workflow navigation.

Plugin developers define the workflow of a plugin. Normally, the workflow gets defined before the screens are implemented (and thus before skinners become active). Skinners are mostly NOT free to choose screen names; most of the screen names are defined by the workflow designer. If for example the *Home* workflow state declares its main screen to be *home*, the skinner must implement a screen file with name *home.xaml* to match that screen name. There might be exceptions from that rule, for example the skinner could show an additional dialog when triggering an action in some underlaying model.

A good rule for skinners is: Use the same screen names as the default skin and implement skins which provide more or less the same functionality as the original ones.

Globalization/Localization (i18n)

When looking through the default skin screen files, I'm sure you have seen some expression like "[Home.Title]" or "[SharesConfig.ChooseShareCategories]". Those expressions are globalized strings. The SkinEngine replaces the globalized string "[Home.Title]" by the localization for the current language. For english, that might be the string "Welcome". You can identify a globalized string by the square brackets and at least one dot in the globalized string name. You can find the resource which will be inserted for that globalized string in the "strings_xx.xml" file for each language. For example, the file "strings_en.xml" contains the english localization strings. In the globalized string [A.B], A is the section of the string and B is the name inside that section.

Default language and string fallback

Each plugin dev is forced to provide english strings for each of the screens/resources that are shown to the user. It makes sense to always test MP2 using the english language.

Additional languages can be added in the same plugin or in different plugins, that doesn't matter, like skin resources, then contents of language files is taken all together, no matter in which files they are defined.

If a string isn't present in the current language, the system tries to load it from the parent language of the current language. For Austria (de-AT), the fallback is german, for example. If the string isn't present in that fallback language, the engine tries the parent language as long as it finds one. The last fallback is always the english language, that's why plugin devs are forced to provide each resource in the english language.

(TODO: Describe how i18n resources are added, establish and describe workflow to keep the resources for all languages in sync)

Creating a new skin

Separation of skin files from plugins

Regardless of talking about the default or any additional skin, skin files can be separated into an arbitrary number of plugins.

In the following sections, we will simplify the situation a bit; we will only talk about one single *skin root directory*. In fact, this logical root directory technically is spread over all enabled plugins in the system and it completely doesn't matter which plugin provides which parts of the skin. Contents of the current skin will be collected by the SkinEngine at startup and mapped into one single name space, where files can be requested with their relative path to the skin's root directory. For example a media file of plugin A might physically be located at *C:\Program files\Team MediaPortal\MP2\MP2-Client\Plugins\A\Skin\default\media\somefileA.png*. A media file of plugin B might have the physical file *C:\Program files\Team MediaPortal\MP2\MP2-Client\Plugins\A\Skin\default\media\somefileB.png*. The files can be referenced by the logical paths *media/somefileA.png* and *media/somefileB.png* all over the skin.

Although it technically doesn't matter how a skin is separated over the plugins, there is a strong recommendation how to organize it: The default skin should be spread over all plugins. Each plugin which defines logical screens must implement those screens at least for the default skin. Additional skins will typically be located in a single plugin which is called the *skin plugin*. For example if you create a "christmas" skin, all skin files of that skin will be typically located in one single plugin of name "Christmas_Skin" (or similar). It is also possible to extend an existing skin by providing new themes or new screen files. Such a skin extension can also be done in its own skin extension plugin. For example if you create a new theme for the default skin, which shows all controls in a light-red tint, that theme could be provided in a plugin of name "default-red-tint" (or similar).

What constitutes a Skin?

Technically, a skin is constituted by a set of files with different kinds:

- *Exactly one skin descriptor (skin.xml)*
- 0-n screen files (*.xaml)
- 0-n Font declaration files
- 0-n Media files (pictures, sound, ...)
- 0-n Style resource files
- others (not specified yet)
- *Theme resources*

Technically, all those files, except those written in italics, can be located both in the skin or in themes. Files located in the theme always override files in the skin with the same relative path (see: "Resource lookup chain"). Sometimes it might be useful to define one resource file in the skin, which then will be overridden in some of the themes. But in most of the cases, resources have a fixed location, either in the skin or in the themes. In the following sections, the file types are explained more detailed and their preferred location will be given.

File locations

In the simplest form, all those files are located relative to the skin root directory (remember that the skin's root directory might be spread over multiple plugins). The skin root directory is located in a plugin (or multiple plugins), where it is declared as a "skin" resource directory by the plugin descriptor (the *plugin.xml* file). The name of the skin root directory has to match the logical skin name. For a description how to write a plugin and how to declare plugin resources, see the Core Services > Plugin Manager page.

Skin descriptor (skin.xml)

The skin descriptor holds metadata for the skin, such as the name, its author, its native screen size, etc. The skin descriptor file has to be located in the skin's root directory. It must be present exactly once. That file should be located in the skin's main plugin. In case of the default skin, it is located in the *Skin Base* plugin.

Screen files

Screen files contain the skin's layout information for special screens, like the placement of the menu and the contents, for all screens. For every screen the system has to have at least one screen file. Screen files are located in the skin's or theme's *screens* subdirectory (preferred location: skin). Complete screen files cannot be edited by Microsoft tools for WPF. We don't have a screenfile editor yet. Instead, they must be coded by hand.

Font files

Font information can be given in several formats *(to be completed)*. Font files are located in the skin's or theme's *fonts* subdirectory. Each font needs its own descriptor file with the file extension *.desc*, also located in the *fonts* directory (preferred location: skin or theme). See existing font definitions for example in the *SkinBase* directory.

Media files

Media files are all files which provide image or sound to the skin. They are referenced from the screen files. Media files are located in the skin's or theme's *media* subdirectory (preferred location: skin or theme).

Style resources

Style resource files contain style information like the appearance of controls, colors and animations. Style resource files are located in the skin's or theme's *styles* subdirectory (preferred location: theme).

HelloWorld Skinfile

To create a typical "Hello World" skinfile, we use a text- or XML-editor. As XAML files are XML files, we need to put an XML header at the top of the file. The root element must always be Screen. Inside the Screen element, any UI element can be placed, we're using a label which shows our hello world text.

```
<?xml version="1.0" encoding="utf-8"?>
<Screen
    xmlns="www.team-mediaportal.com/2008/mpf/directx"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mp_special_workflow="clr-namespace:MediaPortal.UI.SkinEngine.SpecialElements.Workflow"
    mp_special_workflow:WorkflowContext.StateSlot="Root"
    x:Name="MyScreen"
    >
  <Label Content="Hello World"/>
</Screen>
```

As you can see, we use a default XML namespace of _

```
www.team-mediaportal.com/2008/mpf/directx
```

. *This namespace contains all MPF UI elements, and works exactly like the default WPF namesapce*

```
[http://schemas.microsoft.com/winfx/2...l/presentation|http://schemas.microsoft.com/winfx/2006/xaml
/presentation]
```

_ for most of our supported UI elements. The XML namespace declaration _xmlns:x="

```
[http://schemas.microsoft.com/winfx/2006/xaml]
```

", *which defines the _x:* prefix for the current element, is used in the XAML file exactly like it is used in WPF, although we make use of an own implementation of the namespace elements. The declared namespace which is used for the mp_special_workflow prefix is used for the WorkflowContext. StateSlot property which makes the screen remember the last focused control when another workflow state is pushed onto the current state. The default encoding is *utf-8* and should be used on all XAML files.

Resource lookup chain

The skinfile lookup mechanism is designed to make it possible that new skins/themes simply reuse resources from another skin or theme. To reuse resources (files, pictures, workflow descriptor resources etc.) from another skin, simply specify the name of the other skin in your skin descriptor file in the parameter "BasedOnSkin". If you specify a parameter "BasedOnTheme", all skin files are inherited from that theme in the skin given by the "BasedOnSkin" parameter.

```
<?xml version="1.0" encoding="utf-8"?>
<Skin Name="MySkin" Version="1.0">
  <ShortDescription>New skin for MediaPortal 2</ShortDescription>
  <UsageNote></UsageNote>
  <Preview>images\preview.png</Preview>
  <Author>Me</Author>
  <SkinVersion>1.0</SkinVersion>
  <SkinEngine>1.0</SkinEngine>

  <NativeWidth>1280</NativeWidth>
  <NativeHeight>720</NativeHeight>
  <DefaultTheme>Blue</DefaultTheme>
  <BasedOnSkin>OtherSkin</BasedOnSkin>
</Skin>
```

If the "BasedOnSkin" and "BasedOnTheme" parameters are not specified, skins automatically inherit from the default theme which inherits from the default skin.

? Unknown Attachment

The simplest example for reusing resources from another skin is: Just create a new skin which only consists of the skin descriptor in the new skin directory, with only the name and the upper 6 descriptive elements. Nothing else! If you switch to that new skin in the GUI, the new skin will look like the default skin with the default theme. Why? Because the default skin/default theme is the default "parent" resource location. A file the application is searching (e.g. a screen which is to be loaded) is first looked up in the directories of the current theme. If it is not present there, it is looked up in the directories of the current skin. If it is not found there, the inherited theme or skin is used. If the skin doesn't denote a skin or theme it inherits from, the default theme is used to lookup the file. If the file is found somewhere, it is taken from that position. If it was not found, an exception is thrown in the application.

That lookup chain makes it possible to flexibly override single files from the inherited skin in the own skin. That technique is heavily used in the themes of the default skin, for example. See that section for a detailed description.

Usage of the resource lookup chain machinery to achieve simple theming

The different themes of the default skin all look the same except for the colors. How is that achieved? Look into the theme directories. Only the default theme defines all needed theme resources, like ***ButtonStyle***, ***CheckBoxStyle*** etc. All color definitions are outsourced to a file of the name ***Colors.xaml***. The other themes simply override that file with their own colors.
Hint: The simplest way to create a new theme is just to build its directory, create its theme descriptor file and override the _Colors.xaml style file for that skin with colors different from the default theme._

To go on reading

The following sections still need to be written. Until they are available, you can take a look at the current skinfiles in the MediaPortal 2 plugins in the source code.

Deeper look into skin files

(To be continued)

Themes

(To be continued)

HelloWorld Skin

(To be continued)

The default skin

The default skin is the skin which is shown by MediaPortal 2 when no special skin is configured. Its name is *default* and you can find its basics in the *SkinB ase* plugin in folder *Skin/default*. *(to be continued)*