

F.A.P. application step by step.

It's time to explain how to use **F.A.P.** technologies in real world.
(**F.A.P.** for [Fever](#) / [AsWing](#) / [Pixlib](#) frameworks.)

1. [Before starting](#)
2. [Bases](#)
3. [Start your engine](#)
4. [On the road with F.A.P.](#)
 1. [Look and Feel definition](#)
 2. [Application resources](#)
 1. [Localisation / Translation definition](#)
 2. [Bitmaps definition](#)
 3. [Defines Views list](#)
 1. [Menubar View](#)
 2. [Main view](#)
 3. [Main application view](#)
 4. [Events / Commands definition](#)
5. [Compile & Publish](#)
6. [Conclusion & download](#)

1. Before starting

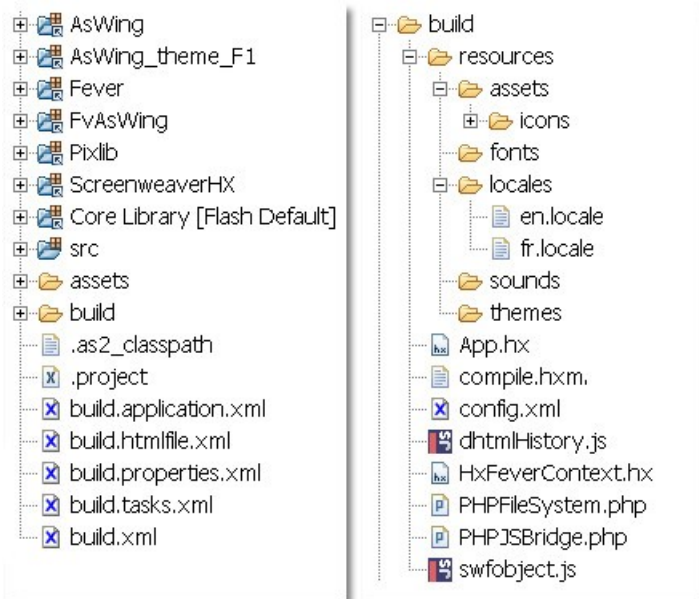
You can read these 2 following articles about [configuration](#) and [Fever first step](#). Really useful for the rest of this tutorial.

Quick note : Tutorial is based on [Eclipse IDE](#) with [FDT](#) plugin for ActionScript edition.

2. Bases

Optional step, but it allow better understanding and organisation between external / internal resources. Check the "*Template project*" from [Google SVN](#) repository and download it to get the structured folders.

This is my structure :



So simply create a new "*Flash Project*" with [FDT](#) and paste the "*Template project*" content. Thus you have the basic and minimal structure to work with [Fever](#).

Before writing the first ActionScript code line, open and customize buildings files :

- **build.xml** : Modify the project name to avoid conflict with others of your project
- **build.properties.xml** : Modify paths using your own configuration
- **build.application.xml** : Modify with your own application setting

3. Start your engine

As you can see, a **Main.as** class and a **template.MyApplication.as** are already written for you. Feel to modify them as you want. (in this tutorial, *template* package is renamed to *demo* package)

Take a look at the **Main.as** class to understand application initialization.

```
public static function main() : Void
{
    FeverDebug.isOn = true;

    Fever.run( new MyApplication(), HaxeContext.getInstance(), true );
}
```

Here we enable the **FeverDebug** logging channel.

Automatically, **SOSTracer** is connected to the **Logging API** and is registered to **FeverDebug** channel. Of course you can easily add your own tracer using **Logger#addLogListener()** method.

```
// add Luminic console
Logger.getInstance().addLogListener( LuminicTracer.getInstance(), FeverDebug.channel );

// add FvAswingTracer component
FvAsWingTracer.connect( FeverDebug.channel );

// Add Firebug console
FireBugTracer.connect( FeverDebug.channel );
```

Now the heart of **Fever**, the magic **Fever#run()** method.

This method defines 3 important points :

- Who is the **main application** class
- In which **context** the application run
- If application use a external **configuration** file

In our example, **MyApplication** is the real main application class (where the application begin). This class must implements **fever.FeverApplication** interface to be a valid argument.

Our application will be run under **ScreenweaverHX** control (and only) so we can specify to use the **HaxeContext** as main context.

Fever has 3 pre implemented contexts :

- **BrowserContext** : work with browser
- **StudioContext** : work with [SWF Studio V3](#) swf2exe application
- **HaxeContext** : work with [Haxe](#) and [ScreenweaverHX](#)

To allow **Cross Context Application** use the magic **ContextSwitcher.init()** instead of any of three previous context in **Fever#run()** method

For more a quick view of **Context API**, you take a look at this presentation article [here](#).

Please read this article ([configuration](#)) about third boolean parameter : [Fever Configuration](#)

Ok... engine is on, let's go now for a concrete application example using **F.A.P.** technologies

4. On the road with F.A.P.

Not really a killer app ;), demo project show you basics use of **F.A.P. frameworks** (Look'n Feel definition, localisation, Bitmap buffering, some [AsWing](#) specific behaviour and more...)
Of course steps defined here are not necessities for all application... they are here just for demo coding ;)

4.1 Look and Feel definition

I use the [AsWing](#) framework for all my application UI now.

2 ways to define Look'n Feel for our application :

- External configuration file (loading external theme definition using the configuration engine)
- Hard coded definition

Here I choose the "hard coded" way because external definition is already view in the configuration ticket ([link](#))

```
1. var o : ThemeLocator = ThemeLocator.getInstance();
2. o.register( 'Office 2003', new Office2003LAF() );
3. o.register( 'WinXP 2003', new WinXPLAF() );
4. o.load( 'WinXP 2003' );
```

Line 1 : retrieve the (singleton) instance of the **ThemeLocator** (themes manager)

Line 2 : register a new theme with name "*Office 2003*" and pass concrete theme implementation using **Office2003LAF** instantiation.

Line 3 : new theme named "*WinXP 2003*"

Line 4 : load the "*WinXP 2003*" theme as startup theme

Ok now to do more complete... I want to have an application background which is in perfect harmony with the Look'n Feel.

So when user change is preferred theme, background must be update too.

To do the job we first define a new background for application :

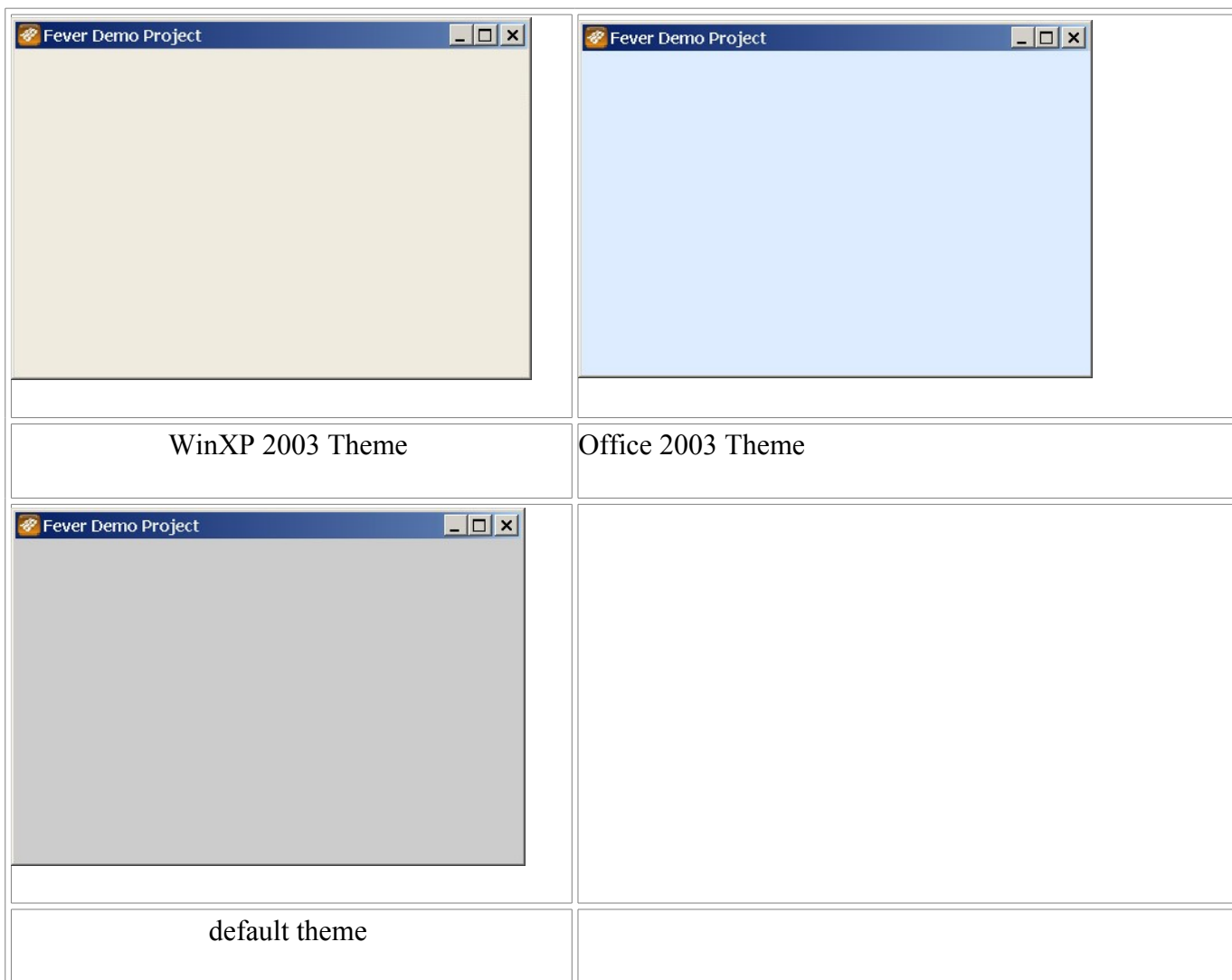
```
1. var color : Number = UIManager.getColor( 'Panel.background' ).getRGB();
2. var bg : Background = new SolidBackGround( color, 100 );
3.
4. FeverStage.getInstance().setBackground( bg );
```

Line 1 : retrieve the color of the **Panel** component in current **Look'n Feel**

Line 2 : create a new **background** (a **SolidBackground** here) with panel color

Line 4 : set the new background as **FeverStage** background

Screenshot with defined background and "WinXP 2003" Look'n Feel"



To keep the color correct when user change the Look'n Feel we have to listen to **ThemeLocator.onChangeEVENT** event type (which is broadcasted by the **ThemeLocator** when a new theme is applied.

```
1. o.addListener( ThemeLocator.onChangeEVENT, this, _onLAFChanged );
```

As you can see, a better implementation is to move the background creation into a private method and to register for event listening before call o.load() method.

Here is a sample source code for correct and better Look'n Feel initialization.

```
private function _configureUI() : Void
{
    var o : ThemeLocator = ThemeLocator.getInstance();
    o.register( 'Office 2003', new Office2003LAF() );
    o.register( 'WinXP 2003', new WinXPLAF() );
    o.addListener( ThemeLocator.onChangeEVENT, this, _onLAFChanged );
    o.load( 'WinXP 2003' );
}

private function _onLAFChanged( event : IEvent ) : Void
{
    var color : Number = UIManager.getColor( 'Panel.background' ).getRGB();
    var bg : Background = new SolidBackGround( color, 100 );
```

```
FeverStage.getInstance().setBackground( bg );  
}
```

4.2 Application resources

We want to develop a multi languages application ("*fr*" and "*en*" for example) and some of our controls (UI) will use icon bitmaps.

So we have to define localisation and translation nodes for multi languages support and to define bitmaps preloading to buffer them for future use.

4.2.1 Localisation / Translation definition

To do the job we first need to edit the "config.xml" file to specify any languages we want. Here we declare just 2 languages "*french*" and "*english*" :

```
<locales>  
  <lang id="fr" label="français"/>  
  <lang id="en" label="english" />  
</locales>
```

Note that **label** attribute is used as label in the **FvLocalisationChooser** component ([FvAsWing](#) framework)

Ok now we have to edit all translation files.

For our example, I only edit the "*english*" one, but the same job must be in "*french*" file too.

We can find basic (and necessary) translation file into the *resources/locales* folder.

I don't write the file content here (too long and not important here), but current content represent the basic translation nodes that [Fever & FvAswing](#) can use in your development. Feel free to remove entire node translation, but as your own risk, if [Fever](#) need it and don't find it, [Fever](#) use a default (*english*) translation; so be sure of what you do ;)

So... we will create a custom translation node for our application; this node will contain all necessities translation test needed in future by application.

```
1. <demo:common>  
2.   <openProjectTips type="string">opens a new project</openProjectTips>  
3.   <quitApplicationTips type="string">quit application</quitApplicationTips>  
4. </demo:common>
```

Line 1 : create a new translation group named "demo:common"

Line 2 : define a new translation node named "openProjectTips" with translation value

Line 3 : //

Line 4 : close the group

We add some new translations later in tutorial, here we have just the base to show how to retrieve easily and really quickly your definition.

2 ways to get your translations in your application

- Strong access to translation **map**
- **Resource** definition

The "strong map access" is really useful when you have little translation node to deal with.

We can access them using the **Localisation#getResource()** static method

```
FeverDebug.DEBUG( Localisation.getResource( 'demo:common.openProjectTips' ) );
```

But when we have to deal with many translations, or just for a better code organization, it can be really great to create a specific **Resources** subclass to wrap all defined nodes.

Resources are automatically connected to the **Localisation API**, so their content are updated when user change the current language.

This the class **DemoResources** (**Resources** subclass)

```
1. import fever.app.local.LocalisationEvent;
2. import fever.core.Resources;
3.
4. /**
5.  * {@code DemoResources } class.
6.  *
7.  * @author Romain Ecarnot
8.  */
9. class demo.DemoResources extends Resources
10. {
11.     //-----
12.     // Private properties
13.     //-----
14.
15.     private static var _instance : DemoResources;
16.
17.
18.     //-----
19.     // Public Properties
20.     //-----
21.
22.     public var openProjectTips : String;
23.     public var quitApplicationTips : String;
24.
25.
26.     //-----
27.     // Public API
28.     //-----
29.
30.     /**
31.      * Returns {@link DemoResources} instance.
32.      */
33.     public static function getInstance() : DemoResources
34.     {
35.         if( !_instance ) _instance = new DemoResources();
36.         return _instance;
37.     }
38.
39.     /**
40.      * Triggered when Localisation language change.
41.      */
42.     public function onLocalisationUpdate(event : LocalisationEvent ) : Void
43.     {
44.         openProjectTips = getTranslation( 'openProjectTips' );
45.         quitApplicationTips = getTranslation( 'quitApplicationTips' );
46.     }
47.
48.
49.     //-----
50.     // Private implementation
```

```

51. //-----
52.
53. /**
54.  * Constructor.
55.  */
56. private function DemoResources()
57. {
58.     super('demo:common');
59. }
60.
61. private function _initDefault() : Void
62. {
63.     openProjectTips = 'opens a new project';
64.     quitApplicationTips = 'quit application';
65. }
66. }

```

Line 22 & 23 : declare 2 public properties to retrieve correct translation ID (property name are not important, here property name are equal to translation node name for better understanding)

Line 42 : call when application language is changed (**Localisation API**)

Line 44 & 45 : retrieve correct translation value for our public properties

Line 58 : call super constructor with node group name

Line 63 & 64 : defines default translation values if localisation failed

So now we can get our translation values with :

```
FeverDebug.DEBUG( DemoResources.getInstance().openProjectTips );
```

4.2.2 Bitmaps definition

We often need bitmaps for our application.

They are 2 main ways to load and use bitmaps in Flash (embedding bitmaps in swf using [SWFMill](#) or loading them externally)

Here we will see how to preload external bitmaps and how bitmaps are **buffered** for future use ([Fever](#) feature)

First step, declare needed bitmaps into configuration file.

```

1. <bitmaps>
2.   <repository>
3.     <asset id="load_icon" url="icons/document-open.png" />
4.     <asset id="locale_icon" url="icons/locale.png" />
5.     <asset id="exit_icon" url="icons/system-log-out.png" />
6.   </repository>
7. </bitmaps>

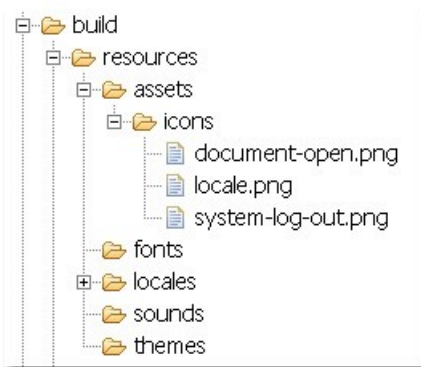
```

Line 2 : Use default bitmap repository (if you want to create new one specify *id* attribute : `<repository id="fx" >`)

Line 3 - 5 : add assets definitions where

- **id** is the registration id of the asset in current repository
- **url** is the relative path of the asset

Bitmaps file structure :



Notice that you don't have to specify full qualified path ('resources/assets/' folder) as this path is defined in configuration file :

```
1 <fever>
2 <resourceURL type="string">resources</resourceURL>
3 <localeURL type="string">locales</localeURL>
4 <fontURL type="string">fonts</fontURL>
5 <themeURL type="string">themes</themeURL>
6 <bitmapURL type="string">assets</bitmapURL>
7 <remotingURL type="string"></remotingURL>
8
```

Here we have defined all bitmaps in "default" repository, feel free to create new one for better organization.

Take a look at this [article](#) to know how to play with [bitmaps repository and libraries](#).

We already define a resource class for our application, use it and register icon registration ID.

Simply define 3 new public properties and initialize them in constructor.

Registration never change, if we want to change icon representation, just change the icon relativ path. (or icon image itself)

DemoResources constructor is now :

```
private function DemoResources()
{
    super( 'demo:common' );

    loadIcon = 'load_icon';
    localIcon = 'locale_icon';
    exitIcon = 'exit_icon';
}
```

Preloading and registration process is automatically done by [Fever](#).

Ok... now let's play

4.3 Defines Views list (Pixlib power)

Firstable, I will create a list of unique ID for all views in application.

Thus I could more easily retrieve and interact with my views. ([Pixlib ViewHelper](#) feature)

Here my list :

```
class demo.ViewList
{
    //-----
    // Public Properties
    //-----

    /** Constant. ID registration for MainUI view. */
    public static var MAIN_VIEW : String = 'MainView';

    /** Constant. ID registration for menubar view. */
    public static var MENUBAR_VIEW : String = 'MenuBarView';

    //-----
    // Private implementation
    //-----

    private function ViewList() {}
}
```

Ok, we have defined 2 different views ID.
Let's create them now.

4.3.1 Menubar View

Menubar view is just a simple **JToolBar** with 3 buttons.
Each button has a custom action and a custom Key accelerator.



I extend the **FvViewHelper** class (abstract implementation of [Pixlib ViewHelper](#) with [AsWing](#) component auto registration).

```
1. class demo.views.MenuBarView extends FvViewHelper implements LocalisationListener
2. {
3.     //-----
4.     // Private properties
5.     //-----
6.
7.     private var _resources : DemoResources;
8.
9.
10.    //-----
11.    // Public API
12.    //-----
13.
14.    public function MenuBarView()
15.    {
16.        super(_createUI(), ViewList.MENUBAR_VIEW );
```

```

17.
18.     _resources = DemoResources.getInstance();
19.
20.     _configureUI();
21.
22.     _registerKeyMapping();
23.     Localisation.addLocalisationListener( this );
24. }
25.
26. //-----
27. // Private implementation
28. //-----
29.
30. private function _createUI() : Container
31. {
32.     return new JPanel( new BorderLayout( 0, 5 ) );
33. }
34. }

```

Line 1 : implements **LocalisationListener** to listen language change

Line 7 : private property to get resources definition

Line 16 : call **super constructor** with

- concrete view component (**UI**)
- View registration **ID**

Line 18 : retrieves resources instance

Line 20 : call **#_configureUI()** to build interface

Line 22 : call **#_registerKeyMapping()** to define keys accelerator

Line 23 : register this class as localisation listener

Line 30 : build concrete component for this view

To keep strong typing in subclass, I overwrite the **FvViewHelper.getView()** method to return the correct component type (here **Container**)

```

/**
 * Returns concrete container.
 */
public function getView() : Container
{
    return Container( view );
}

```

As I say, this view is just a **JToolBar** component with 3 buttons; take a look at UI construction.

```

1. private function _configureUI() : Void
2. {
3.     var gap : Number = 10;
4.
5.     _loadButton = _createButton( _resources.loadIcon, _handleLoadProject, 0, gap );
6.
7.     _localButton = _createButton( _resources.localIcon, _handleLang, gap, gap );
8.     _exitButton = _createButton( _resources.exitIcon, _handleClose, gap, 0 );

```

```

9.
10.  var bar : JToolBar = new JToolBar( "", JToolBar.HORIZONTAL );
11.  bar.append( _loadButton );
12.  bar.append( new JSeparator( JSeparator.VERTICAL ) );
13.
14.  bar.append( _localButton );
15.
16.  getView().append( bar, BorderLayout.WEST );
17.  getView().append( _exitButton, BorderLayout.EAST );
18.  getView().append( new JSeparator( JSeparator.HORIZONTAL ), BorderLayout.SOUTH );
19. }
20.
21. private function _createButton( iconID : String, handler : Function, left : Number, right : Number ) : FvLabelButton
22. {
23.     var button : FvLabelButton = new FvLabelButton( FvBitmapIcon.create( iconID ) );
24.     button.setBorder( new EmptyBorder( null, new Insets( 5, left, 5, right ) ) );
25.     button.addActionListener( handler, this );
26.
27.     return button;
28. }

```

We use **DemoResources** to retrieve icon registration for button, and implement a **#_createButton()** method to centralize creation.

FvLabelButton is used instead of classic **JButton** to disallow button background and border.

Icons are created using **FvBitmapIcon.create()** method which search into repository to find and copy icon representation as [AsWing](#) compliant icon.

Defined now some keyboard shortcuts for buttons

```

1. private function _registerKeyMapping() : Void
2. {
3.     var keyMap : ShortcutMap = ShortcutMap.getApplicationMap();
4.
5.     keyMap.registerCustomType(
6.         FvAsWing.AWSHORTCUT,
7.         new KeyCombo( 'Ctrl+O', Keyboard.onKeyCONTROL, Keyboard.onKeyO ),
8.         _loadButton
9.     );
10.
11.    keyMap.registerCustomType(
12.        FvAsWing.AWSHORTCUT,
13.        new KeyCombo( 'Ctrl+I', Keyboard.onKeyCONTROL, Keyboard.onKeyL ),
14.        _localButton
15.    );
16.
17.    keyMap.registerCustomType(
18.        FvAsWing.AWSHORTCUT,
19.        new KeyCombo( 'Ctrl+w', Keyboard.onKeyCONTROL, Keyboard.onKeyW ),
20.        _exitButton
21.    );
22. }

```

Line 3 : retrieve global keys map (application map is available all the time during application life)

Line 5 : register a new **Shortcut** for [AsWing](#) component (here a **FvLabelButton**)

- Registration type : Here the **FvAsWing.AWSHORTCUT** type constant
- Key shortcut : here a combination of 'Ctrl' + 'o'
- And the target button

Note : I separated key mapping in this sample, but we can register key mapping in **_createButton()** method adding some parameters.

Separation was done here for better code view.

As we implements **LocalisationListener** interface, we have to implement **#onLocalisationUpdate()** method.

When language change, we update the buttons tooltips using resource.

```
public function onLocalisationUpdate( event : LocalisationEvent ) : Void
{
    _loadButton.setToolTipText( _resources.openProjectTips );
    _exitButton.setToolTipText( _resources.quitApplicationTips );
}
```



Handlers now.

Really basic here, we will use classic handler and **Front controller** handler implementation.

```
1. private function _handleLoadProject( source : FvLabelButton ) : Void
2. {
3.
4. FeverController.getInstance().broadcastEvent( new BasicEvent( EventList.onLoadProjectEVENT ) );
5. }
6. private function _handleLang( source : FvLabelButton ) : Void
7. {
8.
9. FeverController.getInstance().broadcastEvent( new BasicEvent( FvAsWingEventList.onLocalisation
10. ChooserEVENT ) );
11. }
12. private function _handleClose( source : FvLabelButton ) : Void
13. {
14.     Fever.application.quit();
15. }
```

Line 3 : broadcast **EventList#onLoadProjectEVENT** event type (**Front Controller API**)

Of course, we have to register Command for this event type, we will see it [later](#).

#onLoadProjectEVENT is a new event type for our application, we have to implement EventList class too.

We use the builtin controller named **FeverController** to dispatch event.

Line 8 : broadcast **FvAsWingEventList.onLocalisationChooserEVENT** event type
This type is a builtin event type defined in **FvAsWingEventList** class, **Command** is already registred for it and opens a **FvLocalisationChooser** dialog.

Line 13 : Simple call to **Fever.application.quit()** method which call the correct context (**Context API**) to close current application.

Our view is finished now,

4.3.2 Main view

Not to complicate this tutorial, main view is just a **JTextArea** component.

```
1. import org.aswing.BorderLayout;
2. import org.aswing.Container;
3. import org.aswing.JPanel;
4. import org.aswing.JScrollPane;
5. import org.aswing.JTextArea;
6.
7. import demo.ViewList;
8.
9. import fvaswing.visual.FvViewHelper;
10.
11. /**
12.  * {@code MainView } class.
13.  */
14. class demo.views.MainView extends FvViewHelper
15. {
16.     //-----
17.     // Private properties
18.     //-----
19.
20.     private var _debugView : JTextArea;
21.
22.
23.     //-----
24.     // Public API
25.     //-----
26.
27.     public function MainView()
28.     {
29.         super( _createUI(), ViewList.MAIN_VIEW );
30.
31.         _configureUI();
32.     }
33.
34.     /**
35.     * Returns concrete container.
36.     */
37.     public function getView() : Container
38.     {
39.         return Container( view );
40.     }
41.
42.     public function appendText( txt : String ) : Void
```

```

43.  {
44.    _debugView.appendText( txt + '\n' );
45.  }
46.
47.
48.  //-----
49.  // Private implementation
50.  //-----
51.
52.  private function _createUI() : Container
53.  {
54.    return new JPanel( new BorderLayout( 0, 5 ) );
55.  }
56.
57.  private function _configureUI() : Void
58.  {
59.    _debugView = new JTextArea();
60.
61.    var scroll : JScrollPane = new JScrollPane( _debugView );
62.
63.    getView().append( _debugView, BorderLayout.CENTER );
64.  }
65. }

```

Note : Look at the constructor and notice the **ViewList.MAIN_VIEW** use.
 We declare a **#appendText()** public method to append some text into textarea.

4.3.3 Append views in main application view.

Our 2 views are build.

As we don't use frame or window system, we have to append views into the main application view.

Use **FvAsWing** to access the main application container.

In our example we simply retrieve the container (JPanel with default BorderLayout) and append view according layout space.

In your **MyApplication._configureUI()** (after Look'n Feel definition)

```

1. var container : Container = FvAsWing.getInstance().getContainer();
2. container.setBorder( new EmptyBorder( null, new Insets( 5,5,5,5 ) ) );
3.
4. container.append( ( new MenuBarView() ).view, BorderLayout.NORTH );
5. container.append( ( new MainView() ).view, BorderLayout.CENTER );

```

Line 1 : retrieve main container

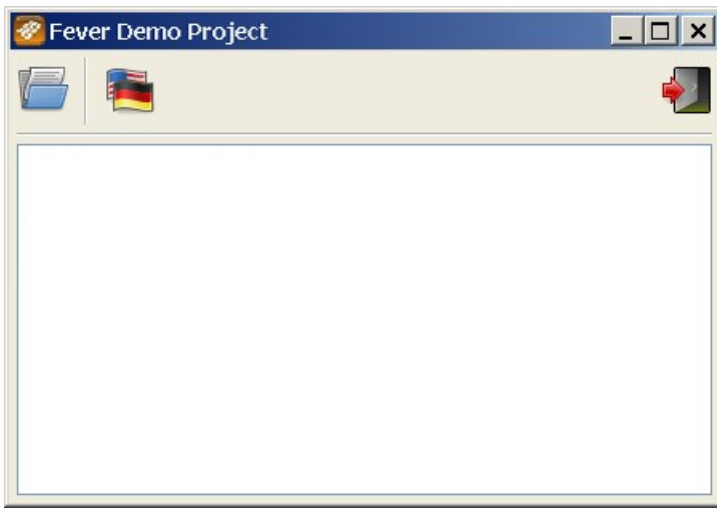
Line 2 : define border for main container

Line 4 : append MenuBarView on north

Line 5 : append MainView on center

That's all.

Here is the final UI screenshot :



4.4 Events / Commands definition (Front controller)

As we see in MenubarView [code](#), the 'loadButton' broadcast event type :

EventList#onLoadProjectEVENT

EventList class act like **ViewList** class, it is just a constant definition class.

```
import com.bourre.events.EventType;

/**
 * Application's event list.
 *
 * @author Romain Ecarnot
 */
class demo.EventList
{
    //-----
    // Public Properties
    //-----

    /** Opens a File open dialog */
    public static var onLoadProjectEVENT : EventType = new EventType( 'onLoadProject' );

    //-----
    // Private implementation
    //-----

    private function EventList() {}
}
```

Front Controller behaviour (summary): when a specific event type is broadcasted by the controller, the controller execute associated **Command**.

So we have to register a command for this **EventList#onLoadProjectEVENT** event type.

We can do it directly into our main application class (here the **MyApplication** class).

```
1. private function _registerCommands() : Void
2. {
3.     FeverController.getInstance().push(
4.         EventList.onLoadProjectEVENT,
5.         new LoadCommand()
6.     );
7. }
```

Line 3 : retrieve **FeverController** instance (**singleton** design)

Line 4 : event type we listen to

Line 5 : Command to execute when event type is broadcasted

Simple isn't it ;)

We don't have to deal with the **FvAsWingEventList.onLocalisationChooserEVENT** event type as it was already defined and associated in **FvAsWing** class.

Let's see our **LoadCommand** class now.

It is an implementation of the **Pixlib Command** interface, so we have to implement a **#execute(event : IEvent)** method, here is a skeleton :

```
import com.bourre.commands.Command;
import com.bourre.events.IEvent;

/**
 * {@code LoadCommand } class.
 */
class demo.commands.LoadCommand implements Command
{
    //-----
    // Public API
    //-----

    /**
     * Constructor.
     */
    public function LoadCommand()
    {
    }

    public function execute( event : IEvent ) : Void
    {
    }
}
}
```

For example, we will open a *File Open* Dialog to select a file.

When user has selected file, we display the filename in Debug area (in **MainView** view)

```

1. public function execute( event : IEvent ) : Void
2. {
3.     var title : String = Localisation.getResource( 'fever:common.open' );
4.
5.     var filter : FileFilter = new FileFilter( 'ActionScript File' );
6.     filter.addExtension( 'as' );
7.
8.     var list : FileFilterList = new FileFilterList( filter );
9.     var newProject : String = Fever.dialog.showFileOpenDialog( ", list, false );
10.
11.     if( newProject != null )
12.     {
13.         var view : MainView;
14.
15.         if( FvViewHelper.isRegistered( ViewList.MAIN_VIEW ) )
16.         {
17.             view = MainView( FvViewHelper.getViewHelper( ViewList.MAIN_VIEW ) );
18.             view.appendText( newProject );
19.         }
20.     }
21. }

```

Line 3 : retrieve a basic translation for dialog title

Line 5-6 : build a file filter

Line 8 : build a file filter list

Line 9 : use the **Cross Context API** to open the *FileOpen* dialog. [Here](#) we use the [ScreenweaverHX](#) dialogs API to open correct result.

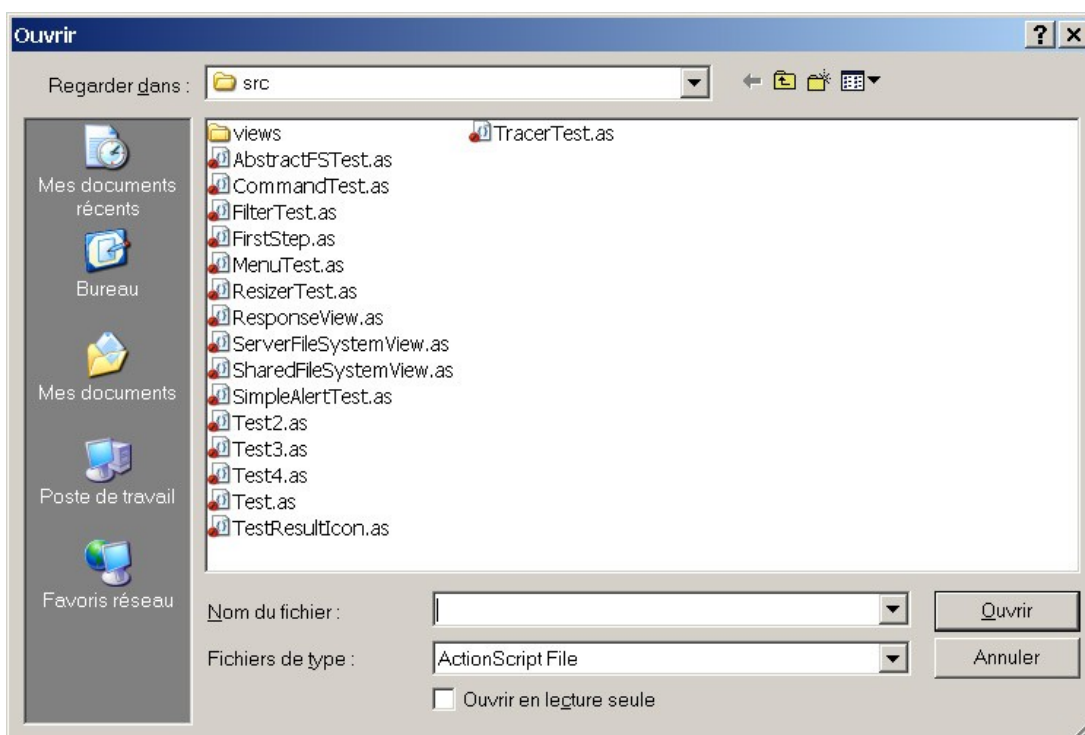
Line 11 : if result is available

Line 16 : checks if a view is registered with the ID : **ViewList.MAIN_VIEW**

Line 17 : if true, retrieve the **MainView** instance

Line 18 : call **#appendText()** wich is a public method of our **MainView** class and pass the dialog result as argument.

Here is a [ScreenweaverHX](#) "FileOpen" dialog screenshot (*fr version*):



Important note about context : If you want to use the **Dialog API** for **browser context**, you have to use the asynschron signature of **API**. And you have to associate **Commands** to :

- **BrowserDialog.onShowFileOpenDialogEVENT** event type
- **BrowserDialog.onShowFileSaveDialogEVENT** event type
- **BrowserDialog.onShowBrowseFolderDialogEVENT** event type

Indeed, we can't stay in synchron mode like other contexts as we open custom dialog UI; we have to listen custom UI dialog result.

For example, if you still using [AsWing](#) as GUI Framework, you can associate builtins commands :

```
FeverController.getInstance().push(  
    BrowserDialog.onShowFileOpenDialogEVENT, new FvShowBrowseFolderDialog()  
);  
  
FeverController.getInstance().push(  
    BrowserDialog.onShowFileSaveDialogEVENT, new FvShowFileSaveDialog()  
);  
  
FeverController.getInstance().push(  
    BrowserDialog.onShowBrowseFolderDialogEVENT, new FvShowBrowseFolderDialog()  
);
```

and call use **Dialog API** like this :

```
1. var newProject : String = Fever.dialog.showFileOpenDialog(  
2.     "  
3.     list,  
4.     false,  
5.     new Delegate( this _onResult )  
6. );
```

As you can notice, we add a **Delegate** as last argument to listen to dialog result.

Don't use callback if you still in synchron mode (**all builtins contexts are synchron except BrowserContext for Dialog API**)

If you want to build a full cross context and use the Dialog API, you have to use asynchron signature for all our calls.

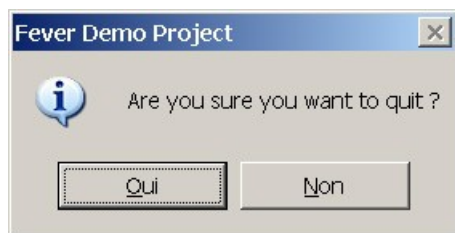
Ok... that's all for this first demo project implementation.

Others menubar actions are automatically process by [Fever](#) :

- Language selector is opened when user click on "*Language button*" (**Fever Front Controller**)



- The application is closed when user click on "*Exit button*"
A warning message is automatically display (according localisation) to prevent user for closing.



5. Compile & Publish

Last step is the compilation time.

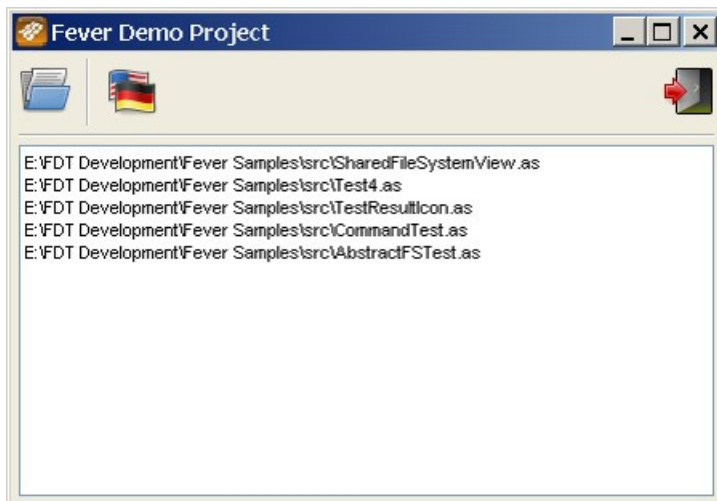
Just use the builtin **Ant Task** named "*buildApplication*". (create a swf file named with **build.application.xml#application.name** property)

Now opens the template "App.n" [Haxe](#) file and change the swf file name, width and height.

```
1. class App
2. {
3.     /**
4.      * Haxe Access point.
5.      */
6.     static function main()
7.     {
8.         swhx.Application.init();
9.
10.        var bridge : HxFeverContext = new HxFeverContext( "DemoProject.swf", 800, 600 );
11.        bridge.start();
12.
13.        swhx.Application.loop();
14.        swhx.Application.cleanup();
15.    }
16. }
```

Let's compile and run... ;)

Application screenshot.



6. Conclusion

You can download source project on : [DemoProject01.zip](#)

Frameworks sources can be downloaded using [Fever Google SVN](#) repository.

Hope this tutorial give you bases to use [Fever](#) & [FvAsWing](#) frameworks.

F.A.P. technologies offer tools making it possible to develop easily, quickly and in a robust way, RIAs and others Flash applications.

To the more will use them and the more rapid will be our development.

This tutorial show basics process to build a an application from scratch; take a look at frameworks sources to discover the real power of **F.A.P.** !

To conclude I want to thanks the **A** and **P** authors ^_^ :

- **Pixlib** framework (Francis Bourre) :: <http://www.osflash.org/pixlib>
 - **PixIOC** : **Pixlib** extension introducing Inversion of Control pattern :: <http://www.osflash.org/pixioc>
- **Aswing** GUI framework (iiley and AsWing developers team) :: <http://www.aswing.org>

At last, sorry for my limited english, the "where is Brian ? " lesson is really very very far ^_^

Romain Ecartot.