Masapi 2.0
XML Specifications

This document describes the specifications for an XML that can be parsed and used by the Masapi project (http://masapi.googlecode.com).

In order to fully understand this document, you must be able to have a good knowledge of AS3, XML and have basically worked with Masapi.

This specification is only available from the version 2.0, revision 165 of the SVN !

Overview

Masapi 2.0 supports three content parts that will be explained in details. Those are :

- *Variables (optional) :* A list of variables that can be used into URL or other properties defines into the files.
- *Files :* A list of files that Masapi will manage.
- *Dependencies (optional) :* A list of dependencies between the files that Masapi will manage.

They must be defined into a UTF-8 XML file in this specified order :

```
<?xml version="1.0" encoding="utf-8" ?>
<application>
	<variables>...</variables>
	<files>...</files>
	<dependencies>...</dependencies>
</application>
```

The last section will describe the complete specifications of each node. All the classes used by Masapi to handle that XML are into the package *ch.capi.net.app*.

Variables

The idea of the variables is to define some values that will be used many times into the XML configuration (and eventually in your application).

A variable is always defined like that :

```
<var name="MY_VARIABLE">my_value</var>
```

To use a variable into your configuration, you must put it between ${...}. You can also reuse variables into other variables :

```
<?xml version="1.0" encoding="utf-8" ?>
<application>
	<variables>
		<var name="ROOT_PATH">./flash</var>
		<var name="PICTURES_PATH">${ROOT_PATH}/pictures</var>
	</variables>
	<files>...</files>
	<dependencies>...</dependencies>
</application>
```

Files

Each node contained in that part represents a file that is represented by the *ApplicationFile* class. For that reason, the parameter *url* has to be defined.

```
<?xml version="1.0" encoding="utf-8" ?>
<application>
	<files>
		<file url="./anim.swf" />
		<file url="./pictures.jpg" />
	</files>
</application>
```

Masapi determines the type of the file using its extension. You may also specify it with the *type* attribute.

```
<?xml version="1.0" encoding="utf-8" ?>
<application>
	<files>
		<file url="./anim.swf" type="swf" />
		<file url="./pictures.jpg" type="binary" />
	</files>
</application>
```

When you use dependencies, or simply if you have to work with a specific file into your code, you'll have to name it. This can be done basicaly using the *name* attribute.

```
<?xml version="1.0" encoding="utf-8" ?>
<application>
	<files>
		<file name="anim" url="./anim.swf" />
		<file name="picture" url="./pictures.jpg" />
	</files>
</application>
```

```
var ctx:IApplicationContext = … //see next chapter
var file:ApplicationFile = ctx.getFile("anim");
```

Note that each name must be unique. If no name is specified, Masapi will automatically generate one.

If you want to load a list of files using priorities, you can specify it with the *priority* attribute :

```
<?xml version="1.0" encoding="utf-8" ?>
<application>
	<files>
		<file name="anim" url="./anim.swf" priority="6" />
		<file name="picture" url="./pictures.jpg" priority="-3" />
	</files>
</application>
```

You can also defines your own attributes that will be put into the *ILoadableFile.properties* property.

Here is an example if you use the variables :

```
<?xml version="1.0" encoding="utf-8" ?>
<application>
	<variables>
		<var name="ROOT_PATH">./flash</var>
		<var name="PICTURES_PATH">${ROOT_PATH}/pictures</var>
		<var name="ANIM">anim.swf</var>
	</variables>
	<files>
		<file name="anim" url="${ANIM}" myProperty="myValue" />
		<file name="picture" url="${PICTURES_PATH}/pictures.jpg" />
	</files>
</application>
```

```
var ctx:IApplicationContext = … //see next chapter
var file:ApplicationFile = ctx.getFile("anim");

trace(file.properties.getValue("myProperty")); //myValue
```

Dependencies

Some file may need other files (text, pictures, …). That's called a dependency in Masapi. To understand how it works, just look at that example :

```
<?xml version="1.0" encoding="utf-8" ?>
<application>
	<files>
		<file name="gallery" url="gallery.swf" />
		<file name="picture1" url="pictures1.jpg" />
		<file name="picture2" url="pictures2.jpg" />
		<file name="picture3" url="pictures3.jpg" />
	</files>
	<dependencies>
		<dependency name="gallery">
			<file name="picture1" />
			<file name="picture2" />
			<file name="picture3" />
		</dependency>
	</dependencies>
</application>
```

This XML defines that the file named 'gallery' needs the files 'picture1', 'picture2' and 'picture3'. Masapi will handle that in a way that once the file gallery is loaded, all its dependencies are already loaded ! And that can be done very easly :

```
var ctx:IApplicationContext = … //see next chapter
var gallery:ApplicationFile = ctx.getFile("gallery");

var appMassLoader:ApplicationMassLoader = new ApplicationMassLoader();
appMassLoader.addFile(gallery); //also add all its dependencies
appMassLoader.start(); //will load 4 files
```

Masapi also supports recursive dependencies ! Here is a more complex example :

```
<?xml version="1.0" encoding="utf-8" ?>
<application>
	<files>
		<file name="page1" url="page1.swf" />
		<file name="page1_title" url="title.txt" />
		<file name="gallery_panel" url="gallery.swf" />
		<file name="gallery_pictures" url="pictures.xml" />
	</files>
	<dependencies>
		<dependency name="gallery_panel">
			<file name="gallery_pictures" />
		</dependency>
		<dependency name="page1">
			<file name="page1_title" />
			<file name="gallery_panel" />
		</dependency>
	</dependencies>
</application>
```

When the *ApplicationMassLoader* class will loads the page1 file, it will first load the file gallery_pictures, then gallery_panel and page1_title and, at the end, the page1 file.

Related classes & interfaces :

ch.capi.net.app.ApplicationFile

Application Context

In an application, you can have to load multiples XML configurations. In order to avoid duplicates, Masapi loads the XML into a specific context. If you want to gain access to it, you may need to name it. That can be simply done in two ways depending of what you need.

One context management

To avoid some work, you can use the dedicated loader of Masapi for your XML configuration. The context name will be defined in the code :

```
var configLoader:ApplicationConfigLoader = ApplicationConfigLoader.create("myContext");
```

That's the easiest and fastest way if you don't have to manage many contexts and configuration files.

```
var configLoader:ApplicationConfigLoader = ApplicationConfigLoader.create("global");
configLoader.addEventListener(Event.COMPLETE, onConfigLoaded);

var request:URLRequest = new URLRequest("masapi-config.xml");
configLoader.load(request);

function onConfigLoaded(evt:Event):void
{
   var context:IApplicationContext = configLoader.applicationContext.
   //...
}
```

Many contexts management

The second approach is to define the name of the context into the XML. If you use this feature, you'll need to load manually your XML configuration.

```
<?xml version="1.0" encoding="utf-8" ?>
<application contextName="myContext">
      <variables>...</variables>
      <files>...</files>
      <dependencies>...</dependencies>
</application>
```

Once loaded, parse your XML like that :

```
var myContext:IApplicationContext = ApplicationFileParser.parse(xmlSource);
```

Note that all the context names must be unique, otherwise an error will be thrown !

```
var xmlLoader:URLLoader = new URLLoader();
xmlLoader.dataFormat = URLLoaderDataFormat.TEXT;
xmlLoader.addEventListener(Event.COMPLETE, onXmlLoaded);
xmlLoader.load(new URLRequest("masapi-config.xml");

function onXmlLoaded(evt:Event):void
{
   var source:String = xmlLoader.data;
   var ctx:IApplicationContext = ApplicationFileParser.parse(source);
   //...
}
```

Retrieve a defined context

Once defined, you can retrieve a context at any time using the *ApplicationContextRegisterer* class :

```
ApplicationContextRegisterer.get("myContext");
```

Related classes & interfaces :

ch.capi.net.app.ApplicationConfigLoader
ch.capi.net.app.ApplicationFileParser
ch.capi.net.app.ApplicationContext
ch.capi.net.app.IApplicationContext
ch.capi.net.app.ApplicationContextRegisterer

Application MassLoader

Once you have an application context, you can use the *ApplicationMassLoader* class to load your files.

Loading all the files

Basically, you'll do something like that :

```
var massLoader:ApplicationMassLoader = new ApplicationMassLoader();
//register listeners...

var configLoader:ApplicationConfigLoader = ApplicationConfigLoader.create("global");
configLoader.addEventListener(Event.COMPLETE, onConfigLoaded);

var request:URLRequest = new URLRequest("masapi-config.xml");
configLoader.load(request);

function onConfigLoaded(evt:Event):void
{
   var context:IApplicationContext = configLoader.applicationContext.
   massLoader.addAll(context);
   massLoader.start();
}
```

There is also an easier way, using directly the *ApplicationConfigLoader.loadAll()* method :

```
var configLoader:ApplicationConfigLoader = ApplicationConfigLoader.create("global");
configLoader.addEventListener(Event.COMPLETE, onConfigLoaded);

var request:URLRequest = new URLRequest("masapi-config.xml");
var massLoader:ApplicationMassLoader = configLoader.loadAll(request);
//register listeners...
```

Both methods will put all the files into the MassLoader and start it. Once the MassLoader dispatches the *Event.COMPLETE* event, all the files defined into your XML will be available !

Also note that in both ways, if you defined specified priorities in the XML configuration, they will be respected !

Loading one file (and its dependencies)
The method is the same as above :

```
var massLoader:ApplicationMassLoader = new ApplicationMassLoader();
//register listeners...

var configLoader:ApplicationConfigLoader = ApplicationConfigLoader.create("global");
configLoader.addEventListener(Event.COMPLETE, onConfigLoaded);

var request:URLRequest = new URLRequest("masapi-config.xml");
configLoader.load(request);

function onConfigLoaded(evt:Event):void
{
   var context:IApplicationContext = configLoader.applicationContext.
   var file:ApplicationFile = context.getFile("myFile");
   massLoader.addFile(file);
   massLoader.start();
}
```

The *ApplicationMassLoader.addFile()* method will put the specified ApplicationFile into the loading queue and all the files it needs in a higher priority.

Related classes & interfaces :

ch.capi.net.app.ApplicationMassLoader
ch.capi.net.app.ApplicationFile

XML Specifications

Here are the specifications that describes all the features supported by Masapi.

Path to node : Defines the node path. For example for /application/files/file it means that sort of structure :

```
<application>
      <files>
            <file … />
      </files>
</application>
```

Attribute name : The name of the attribute. If the attribute name is not specified, the entire node is considered.
Description : Short description of the node/attribute.
V. means 'Variables support'. It defines if the variables put into the value will be interpreted or not. If it is filled on a node, then it means that the node can have a content.
M. means 'Mandatory'. If yes, then the attibute/node must appears on the XML. If nothing is defined, then it's No.

Path to node	Attribute name	Description	V.	M.
/application		Application root node		Yes
/application	contextName	Defines the context name (manual load of XML)	No	
/application/variables		Variables container.		
/application/variables/var		Defines a variable. Content supported.	Yes	
/application/variables/var	name	Defines the variable name.	No	Yes
/application/files		Files container		Yes
/application/files/file		Defines a file. No content supported.		
/application/files/file	url	Defines the file URL.	Yes	Yes
/application/files/file	name	Defines the file name. If not defined, it will automatically be generated.	No	
/application/files/file	type	Defines the type of the file. The types can be : swf, binary, text, variables, sound, stream. See the *LoadableFileType* class.	No	
/application/files/file	global	Defines the file as global. This is a boolean value (true, false, 1, 0). Global files can be retrieved by the *IApplicationContext* interface. The value is false by default.	No	
/application/files/file	priority	Defines the file priority. This must be an Integer (positive or negative) value. If not defined, this value is zero.	No	
/application/files/file	useCache	Defines if Masapi must use the cache for this file. This is a boolean value (true, false, 1, 0). If not defined, true by default.	No	
/application/files/file	netState	Defines the NetState. The value can be 'online', 'offline' or 'dynamic'. If not defined, 'dynamic' by default. See the *NetState* class.	No	
/application/files/file	virtualBytesTotal	Defines the virtual bytes total. This is a positive Integer value. It is used to calculate the percentage before the bytesTotal value is known.	No	

/application/files/file	appDomain	Defines the application domain. The values can be 'create', 'child', 'current' or 'none'. If not defined, 'none' by default. See the *DomainUtils* class.	No	
/application/files/file	secDomain	Defines the security domain. The values can be 'current' or 'none'. If not defined, 'none' by default. See the *DomainUtils* class.	No	
/application/files/file	requestMethod	Defines the request method. The values can be 'GET' or 'POST'. If not defined, 'GET' by default.	No	
/application/files/file	requestData	Defines the data of the request. The value must be formated as for URLVariables. Thoses variables will be sent using the requestMethod value.	Yes	
/application/files/file	*	Any other attribute can be defined and will be put into the properties of the *ApplicationFile* class. See the *IProperties.getUpdateValue()* method.	Yes	
/application/dependencies		Dependencies container		
/application/dependencies /dependency or /file		Defines a dependency		
/application/dependencies /dependency or /file	name	Defines the name of the file to express its dependencies. The file has to be defined and named info /application/files.	No	Yes
/application/dependencies /dependency/file or /file/file		Defines a dependency of the parent file, eq. that file must be loaded before the parent is loaded.		
/application/dependencies /dependency/file or /file/file	name	Defines the name of the file.	No	Yes
/application/dependencies /group		Defines a group. A group is a virtual dependency that is needed to do groups with files.		
/application/dependencies /group	name	Defines the name of the group.	No	