GradleFx Documentation

Release 0.6.3

GradleFx

September 08, 2013

CONTENTS

Contents:

CHAPTER

ONE

BASIC SETUP

1.1 Requirements

- Gradle v1.0

1.2 Using the plugin in your project

To use the plugin in your project, you'll have to add the following to your build.gradle file:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'org.gradlefx', name: 'gradlefx', version: '0.5'
    }
}

apply plugin: 'gradlefx'
```

Make sure that the buildscript structure is at the top of your build file.

1.3 Setting up the Flex/Air SDK

GradleFx gives you several options to specify the Flex/AIR SDK:

1. set the FLEX_HOME environment variable (convention), this should point to your Flex/AIR SDK installation.
2. set the flexHome convention property to the location of your Flex/AIR SDK

   ```
   flexHome = "C:/my/path/to/the/flex/sdk"
   ```

3. specify the Flex/AIR SDK as a dependency. See *Flex/AIR SDK Auto Install*

1.4 Defining the project type

Every project should define its type, this can be one of the following:

swc: a library project of which the sources will be packaged into a swc file
swf: a Flex web project of which the sources will be packaged into a swf file.
air: a Flex web project of which the sources will be packaged into a air file.

example project type definition:

```
type = 'swc'
```

CHAPTER

TWO

FLEX/AIR SDK AUTO INSTALL

GradleFx gives you the option to automatically download and install the Flex/AIR SDK. You can do this by specifying either of them as a dependency. This mechanism supports both the Adobe and the Apache Flex SDK.

2.1 Overview

When you specify the SDK's you'll always have to use a packaged SDK. The supported archive formats are zip, tar.gz and tbz2.

What basically happens when you declare the dependency is this:

1. GradleFx will determine the install location of the SDK. By convention it will create an SDK specific directory in the %GRADLE_USER_HOME%/gradlefx/sdks directory. The name of the SDK specific directory is a hash of the downloaded sdk archive location.
2. When the SDK isn't yet installed GradleFx will install it.
3. Once installed it will assign the install location to the flexHome convention property.

GradleFx will always install the AIR SDK in the same directory as the Flex SDK.

Note: A sample project which uses the auto-install feature can be found here: Auto-install sample

---

2.2 Dependency types

There are a couple of ways to specify the SDK's as dependencies.

2.2.1 Maven/Ivy Dependency

If you have deployed the SDK archives to a Maven/Ivy repository then you can specify them like this:

```
dependencies {
        flexSDK group: 'org.apache', name: 'apache-flex-sdk', version: '4.9.0', ext: 'zip'
        airSDK  group: 'com.adobe', name: 'AdobeAIRSDK', version: '3.4', ext: 'zip'
    }
```

2.2.2 URL-based Dependency

You can also specify the SDK by referencing a URL. To do this you need to define custom Ivy URL Resolvers. For example for the Apache Flex SDK this would be something like this:

```
repositories {
        add(new org.apache.ivy.plugins.resolver.URLResolver()) {
                name = 'Apache'
                // pattern for url http://apache.cu.be/flex/4.9.0/binaries/apache-flex-sdk-4.9.0-bin
                addArtifactPattern 'http://apache.cu.be/flex/4.9.0/binaries/[module]-[revision]-bin.
        }
}
```

Always make sure to replace the artifact name, version and extension type with [module], [revision] and [ext] in the pattern. Once you've defined the pattern you can define the dependencies like this:

```
dependencies {
        flexSDK group: 'org.apache', name: 'apache-flex-sdk', version: '4.9.0', ext: 'zip'
        airSDK  group: 'com.adobe', name: 'AdobeAIRSDK', version: '3.4', ext: 'zip'
    }
```

2.2.3 File-based dependency

And the last option is to specify the SDK's as file-based dependencies. This can be done as follows:

```
dependencies {
        flexSDK files('C:/sdks/flex-4.6-sdk.zip')
        airSDK  files('C:/sdks/air-3.4-sdk.zip')
    }
```

2.3 Apache Flex SDK dependencies

As you may probably know the Apache Flex SDK requires some dependencies that aren't included in the SDK archive. GradleFx handles the installation of these dependencies for you. During the installation some prompts will be shown to accept some licenses. When you've made sure you read the licenses, you can turn the prompts off (e.g. for a continuous integration build) like this:

```
sdkAutoInstall {
        showPrompts = false
}
```

CHAPTER

THREE

PROPERTIES/CONVENTIONS

The GradleFx plugin provides some properties you can set in your build script. Most of them are using conventions, so you'll only need to specify them if you want to use your own values.

The following sections describe the properties you can/have to specify in your build script(required means whether you have to specify it yourself):

3.1 Standard Properties

Property Name	Convention	Required	Description
gradleFxUserHomeDir	%GRADLE_USER_HOME%/gradleFx	false	The location where GradleFx will store GradleFx specific files (e.g. installed SDK's)
flexHome	FLEX_HOME environment var	false	The location of your Flex SDK
type	n/a	true	Whether this is a library project or an application. Possible values: 'swc' or 'swf'
srcDirs	['src/main/actionscript']	false	An array of source directories
resourceDirs	['src/main/resources']	false	An array of resource directories (used in the copyresources task, or included in the SWC for library projects)
testDirs	['src/test/actionscript']	false	An array of test source directories
testResourceDirs	['src/test/resources']	false	An array of test resource directories
includeClasses	null	false	Equivalent of the include-classes compiler option. Accepts a list of classnames
includeSources	null	false	Equivalent of the include-sources compiler option. Accepts a list of classfiles and/or directories.
frameworkLinkage	'external' for swc projects, 'rsl' for swf projects and 'none' for pure as projects	false	How the Flex framework will be linked in the project: "external", "rsl", "merged" or "none"
useDebugRSLSwfs	false	false	Whether to use the debug framework rsl's when frameworkLinkage is rsl
additionalCompilerOptions	[]	false	Additional compiler options you want to specify to the compc or mxmlc compiler. Can be like ['-player-version=10', '-strict=false']
fatSwc	null	false	When set to true the asdoc information will be embedded into the swc so that Adobe Flash Builder can show the documentation
localeDir	'src/main/locale'	false	Defines the directory in which locale folders are located like en_US etc.
locales	[]	false	The locales used by your application. Can be something like ['en_US', 'nl_BE']
mainClass	'Main'	false	This property is required for the mxmlc compiler. It defines the main class of your application. You can specify your own custom file like 'org/myproject/MyApplication.mxml' or 'org.myproject.MyApplication'
output	${project.name}	false	This is the name of the swc/swf that will be generated by the compile task
jvmArguments	[]	false	You can use this property to specify jvm arguments which are used during the compile task. Only one jvm argument per array item: e.g. jvmArguments = ['-Xmx1024m','-Xms512m']
playerVersion	'10.0'	false	Defines the flash player version
htmlWrapper	complex property	false	This is a complex property which contains properties for the createHtmlWrapper task
flexUnit	complex property	false	This is a complex property which contains properties for the flexUnit task
air	complex property	false	This is a complex property which contains properties for AIR projects
asdoc	complex property	false	This is a complex property which contains properties for the asdoc task
sdkAutoInstall	complex property	false	This is a complex property which contains properties for the SDK auto install feature

Note: All the available compiler options for the mxmlc and compc compiler are available here Compc options , Mxmlc options

3.2 Complex properties

3.2.1 air

Property Name	Convention	Required	Description
keystore	"${project.name}.p12"	false	The name of the certificate which will be used to sign the air package. Uses the project name by convention.
storepass	null	true	The password of the certificate
applicationDescriptor	"src/main/actionscript/${project.name}-app.xml"	false	The location of the air descriptor file. Uses the project name by convention for this file.
includeFileTrees	null	false	A list of FileTree objects which reference the files to include into the AIR package, like application icons which are specified in your application descriptor. Can look like this: air.includeFileTrees = [fileTree(dir: 'src/main/actionscript/', include: 'assets/appIcon.png')]

3.2.2 htmlWrapper

Property Name	Convention	Required	Description
title	project.description	false	The title of the html page
file	"${project.name}.html"	false	Name of the html file
percentHeight	'100'	false	Height of the swf in the html page
percentWidth	'100'	false	Width of the swf in the html page
application	project.name	false	Name of the swf object in the HTML wrapper
swf	project.name	false	The name of the swf that is embedded in the HTML page. The '.swf' extension is added automatically, so you don't need to specify it.
history	'true'	false	Set to true for deeplinking support.
output	project.buildDir	false	Directory in which the html wrapper will be generated.
expressInstall	'true'	false	use express install
versionDetection	'true'	false	use version detection
source	null	false	The relative path to your custom html template
tokenReplacements	[application: wrapper.application, percentHeight: "$wrapper.percentHeight%", percentWidth: "$wrapper.percentWidth%", swf: wrapper.swf, title: wrapper.title]	false	A map of tokens which will be replaced in your custom template. The keys have to be specified as ${key} in your template

3.2.3 flexUnit

(Since GradleFx uses the FlexUnit ant tasks it also uses the same properties, more information about the properties specified in this table can be found in the "Property Descriptions" section on this page: http://docs.flexunit.org/index.php?title=Ant_Task)

Property Name	Convention	Required	Description
player	'flash'	false	Whether to execute the test SWF against the Flash Player or ADL. See the "Property Descriptions" section on this page for more information: http://docs.flexunit.org/index.php?title=Ant_Task
command	FLASH_PLAYER_EXE environment variable	false	The path to the Flash player executable which will be used to run the tests
swf	"${project.buildDirName}/${testOutput}.swf"	false	Location of the generated swf files which runs the tests
toDir	"${project.buildDirName}/reports"	false	Directory to which the test result reports are written
workingDir	project.path	false	Directory to which the task should copy the resources created during compilation.
haltonfailure	'false'	false	Whether the execution of the tests should stop once a test has failed
verbose	'false'	false	Whether the tasks should output information about the test results
localTrusted	'true'	false	The path specified in the 'swf' property is added to the local FlashPlayer Trust when this property is set to true.
port	'1024'	false	On which port the task should listen for test results
buffer	'262144'	false	Data buffer size (in bytes) for incoming communication from the Flash movie to the task. Default should in general be enough, you could possibly increase this if your tests have lots of failures/errors.
timeout	'60000'	false	How long (in milliseconds) the task waits for a connection with the Flash player
failureproperty	'flexUnitFailed'	false	If a test fails, this property will be set to true
headless	'false'	false	Allows the task to run headless when set to true.
display	'99'	false	The base display number used by Xvnc when running in headless mode.
includes	['**/*Test.as']	false	Defines which test classes are executed when running the tests
excludes	[]	false	Defines which test classes are excluded from execution when running the tests

3.2.4 asdoc

Property Name	Convention	Required	Description
outputDir	'doc'	false	The directory in which the asdoc documentation will be created
additionalASDocOptions	[]	false	Additional options for the asdoc compiler.

3.2.5 sdkAutoInstall

Property Name	Convention	Required	Description
showPrompts	true	false	Whether to show prompts during the installation or let it run in full auto mode. Make sure you agree with all the licenses before turning this off

Note: All the available asdoc options (for Flex 4.6) can be found here: asdoc compiler options

3.3 Example usage (build.gradle)

```
buildscript {
    repositories {
        mavenLocal()
    }
    dependencies {
        classpath group: 'org.gradlefx', name: 'gradlefx', version: '0.5'
    }
}

apply plugin: 'gradlefx'

flexHome = System.getenv()['FLEX_SDK_LOCATION'] //take a custom environment variable which contains t

srcDirs = ['/src/main/flex']

additionalCompilerOptions = [
  '-player-version=10',
  '-strict=false'
]

htmlWrapper {
            title           'My Page Title'
            percentHeight   '80'
            percentWidth    '80'
    }
```

CHAPTER

FOUR

DEPENDENCY MANAGEMENT

4.1 Overview

The GradleFx plugin adds the following configurations to your project:

- **merged**: This configuration can be used for dependencies that should be merged in the SWC/SWF. Same as -compiler.library-path
- **internal**: The dependency content will be merged in the SWC/SWF. Same as -compiler.include-libraries
- **external**: The dependency won't be included in the SWC/SWF. Same as -compiler.external-library-path
- **rsl**: The SWF will have a reference to load the dependency at runtime. Same as -runtime-shared-library-path
- **test**: This is for dependencies used in unit tests
- **theme**: The theme that will be used by the application. Same as -theme

You can specify your dependencies like this:

```
dependencies {
   external group: 'org.springextensions.actionscript', name: 'spring-actionscript-core', version: '1
   external group: 'org.as3commons', name: 'as3commons-collections', version: '1.1', ext: 'swc'
   external group: 'org.as3commons', name: 'as3commons-eventbus', version: '1.1', ext: 'swc'

   merged group: 'org.graniteds', name: 'granite-swc', version: '2.2.0.SP1', ext: 'swc'
   merged group: 'org.graniteds', name: 'granite-essentials-swc', version: '2.2.0.SP1', ext: 'swc'

   theme group: 'my.organization', name: 'fancy-theme', version: '1.0', ext: 'swc'
}
```

4.2 Project Lib Dependencies

You can also add dependencies to other projects, as described here in the Gradle documentation:
http://www.gradle.org/current/docs/userguide/userguide_single.html#sec:project_jar_dependencies

CHAPTER

FIVE

TASKS

5.1 Overview

The GradleFx plugin adds the following tasks to your project:

Task name	Depends on	Description
clean	n/a	Deletes the build directory
compileFlex	copyre-sources	Creates a swc or swf file from your code. The 'type' property defines the type of file
package	compile	Packages the generated swf file into an .air package
copyresources	n/a	Copies the resources from the source 'resources' directory to the build directory
publish	n/a	Copies the files from the build directory to the publish directory.
createHtmlWrap-per	n/a	Creates an HTML wrapper for the project's swf
test	testCompile	Runs the FlexUnit tests
asdoc	testCompile	Creates asdoc documentation for your sources

The Flashbuilder plugin adds the following tasks to your project:

Task name	Depends on	Description
flashbuilder	n/a	Creates the Adobe Flash Builder project files
flashbuilderClean	n/a	Deletes the Adobe Flash Builder project files

The Scaffold plugin adds the following tasks to your project:

Task name	Depends on	Description
scaffold	n/a	Generates directory structure and main application class

5.2 Adding additional logic

Sometimes you may want to add custom logic right after or before a task has been executed. If you want to add some logging before or after the compile task, you can just do this:

```
compile.doFirst {
            println "this gets printed before the compile task starts"
}

    compile.doLast {
            println "this gets printed after the compile task has been completed"
    }
```

CHAPTER

SIX

AIR

This page describes how you need to configure your AIR project. Only a few things are needed for this.

Note: There's a working example available in the GradleFx examples project: https://github.com/GradleFx/GradleFx-Examples/tree/master/air-single-project

6.1 Project type

First you'll need to specify the project type, which in this case is 'air'. You do this as follows:

```
type = 'air'
```

6.2 AIR descriptor file

Then you'll need an AIR descriptor file (like in every AIR project). If you give this file the same name as your project and put it in the default source directory (src/main/actionscript) then you don't have to configure anything because this is the convention. If you want to deviate from this convention you can specify the location like this:

```
air {
            applicationDescriptor   'src/main/resources/airdescriptor.xml'
    }
```

6.3 Certificate

Then you'll need a certificate to sign the AIR package. This certificate has to be a *.p12 file. GradleFx uses the project name for the certificate by convention, so if your certificate is located at the root of your project and has a %myprojectname%.p12 filename; then you don't have to configure anything. If you want to deviate from this convention, then you can do this by overriding the air.keystore property:

```
air {
            keystore        'certificate.p12'
    }
```

You also need to specify the password for the certificate. This property is required. You can specify this as follows:

```
air {
            storepass       'mypassword'
    }
```

If you don't want to put the password in the build file then you can use the properties system of Gradle, see the Gradle documentation for more information about this: http://www.gradle.org/docs/current/userguide/tutorial_this_and_that.html#sec:gradle_properties_and_system_properties

6.4 Adding files to the AIR package

In most cases you will want to add some files to your AIR package, like application icons which are being specified in your application descriptor like this:

```
<icon>
    <image32x32>assets/appIcon.png</image32x32>
</icon>
```

Only specifying those icons in your application descriptor won't do it for the compiler, so you need to provide them to it. With GradleFx you can do that with the includeFileTrees property, which looks like this:

```
air {
            includeFileTrees = [
                    fileTree(dir: 'src/main/actionscript/', include: 'assets/appIcon.png')
            ]
    }
```

You have to make sure that the 'include' part always has the same name as the one specified in your application descriptor, otherwise the compiler won't recognize it. The fileTree also accepts patterns and multiple includes, more info about this can be found in the Gradle documentation: http://gradle.org/docs/current/userguide/working_with_files.html

CHAPTER

SEVEN

FLEXUNIT

GradleFx supports automatically running tests written with FlexUnit 4.1.

7.1 Setting up testing in GradleFx

First you need to specify the FlexUnit dependencies. You can download the required FlexUnit libraries from their site and then deploy them on your repository (recommended) or use file-based dependencies. Once you've done that you have to define them as dependencies in your build file.

1. When you have deployed the artifacts on your own repository:

```
dependencies {
    test group: 'org.flexunit', name: 'flexunit-tasks', version: '4.1.0-8', ext: 'swc'
    test group: 'org.flexunit', name: 'flexunit', version: '4.1.0-8', ext: 'swc'
    test group: 'org.flexunit', name: 'flexunit-cilistener', version: '4.1.0-8', ext: 'swc'
}
```

2. When you have FlexUnit installed on your machine:

```
def flexunitHome = System.getenv()['FLEXUNIT_HOME']  //FLEXUNIT_HOME is an environment variable
dependencies {
    test files("${flexunitHome}/flexunit-4.1.0-8-flex_4.1.0.16076.swc",
               "${flexunitHome}/flexUnitTasks-4.1.0-8.jar",
               "${flexunitHome}/flexunit-cilistener-4.1.0-8-4.1.0.16076.swc")
}
```

Then you'll need to specify the location of the Flash Player executable. GradleFx uses the FLASH_PLAYER_EXE environment variable by convention which should contain the path to the executable. If you don't want to use this environment variable you can override this with the 'flexunit.command' property. You can download the executable from here (these links may get out of date, look for the Flash Player standalone/projector builds on the Adobe site):

- For Windows
- For Mac
- For Linux

And that's basically it in terms of setup when you follow the following conventions:

- Use src/test/actionscript as the source directory for your test classes.
- Use src/test/resources as the directory for your test resources.
- You end all your test class names with "Test.as"

GradleFx will by convention execute all the *Test.as classes in the test source directory when running the tests.

7.2 Running the tests

You can run the FlexUnit tests by executing the "gradle test" command on the command-line.

7.3 Customization

7.3.1 Changing the source/resource directories

You can change these directories by specifying the following properties like this:

```
testDirs = ['src/testflex']
testResourceDirs = ['src/testresources']
```

7.3.2 Include/Exclude test classes

You can include or exclude test classes which are being run by specifying a pattern to some GradleFx properties. To specify the includes you can use the flexUnit.includes property:

```
flexUnit {
            includes = ['**/Test*.as'] //will include all actionscript classes which start with 'Test
    }
```

To specify the excludes you can use the flexUnit.excludes property:

```
flexUnit {
            excludes = ['**/*IntegrationTest.as']
    }
```

7.3.3 Other customizations

There are a lot more properties available on flexUnit.*, all these can be found on the properties description page.

CHAPTER

EIGHT

HTML WRAPPER

GradleFx allows you to create a html wrapper for your application by using the createHtmlWrapper task and the htmlWrapper convention properties.

8.1 Usage

8.1.1 Execution

You can create the html wrapper files without having to specify any htmlWrapper convention properties. Just execute the createHtmlWrapper task like this and it will use the conventions:

```
>gradle createHtmlWrapper
```

8.1.2 Customization

You can customize the conventions by overriding the htmlWrapper properties, like this:

```
htmlWrapper {
            title           'My Page Title'
            percentHeight   '80'
            percentWidth    '80'
    }
```

Note: For a full list of htmlWrapper properties, visit the properties section: *Properties/Conventions*

You can also provide your own html page which contains replaceable tokens. This can be done with the help of the htmlWrapper.source and htmlWrapper.tokenReplacements properties. source is the relative path to an existing HTML-file that can be provided as a template instead of using the default one. If the property isn't provided, the template will be generated with the default html file.

tokenReplacements is map of replacements for tokens in the provided source file. If the template contains the token ${swf}, it'll be replaced with 'example' if this property contains a [swf:example] mapping. If source isn't specified, this property will be ignored.

You can use this as follows:

```
htmlWrapper {
            source          'myCustomTemplate.html'
            tokenReplacements [swf:example]
    }
```

CHAPTER

NINE

ASDOC

GradleFx has support for generating asdoc documentation for your swc-based projects.

9.1 How to use it

No specific configuration is needed for this, you can simply execute the "gradle asdoc" command and it will create a doc folder in your project which will contain the html documentation files.

9.1.1 Creating a fat swc

A fat swc is a swc file which has asdoc information embedded in it so that Adobe Flash Builder can show the documentation while you're working in it. GradleFx has a handly property for this which, when turned on, will always create a fat swc when you compile your project. This property can be set like this:

```
fatSwc = true
```

9.1.2 Customizing the asdoc generation

GradleFx also provides some properties which can be used to customize the asdoc generation. One of them is the asdoc.outputDir property, which allows you to specify a different destination directory for the asdoc documentation. This property can be used as follows:

```
asdoc {
            outputDir       'documentation' //will create the documentation in the %projectdir%/docur
    }
```

Another property which allows the most customization is the asdoc.additionalASDocOptions property. It can be used like the additionalCompilerOptions, but this one accepts asdoc compiler options. These options can be found here (for Flex 4.6): asDoc compiler options

The property can be used as follows:

```
asdoc {
            additionalASDocOptions = [
                    '-strict=false',
                    '-left-frameset-width 200'
            ]
    }
```

CHAPTER

TEN

LOCALIZATION

GradleFx provides an easy way to specify locales instead of having to specify the compiler arguments. The two convention properties of importance are:

- **localeDir**: This defines the directory in which locale folders are located (relative from the project root). The convention here is 'src/main/locale'
- **locales**: Defines a list of locales used by your application, like en_US, nl_BE, etc. This property has no default.

Let's say you want to support the en_GB and nl_BE locales. Then you could have the following directory structure:

- %PROJECT_ROOT%/src/main/locale/en_GB/resources.properties
- %PROJECT_ROOT%/src/main/locale/nl_BE/resources.properties

Because 'src/main/locale' is already the default value for the localeDir property you only have to specify the locales, like this:

```
locales = ['en_GB', 'nl_BE']
```

You can also change the default value of the localeDir in case you don't want to follow the convention like this:

```
localeDir = 'locales' //directory structure will then look like this: %PROJECT_ROOT%/locales/en_GB
```

CHAPTER

ELEVEN

IDE PLUGIN

This feature mimics the behavior of the 'eclipse', 'idea', etc. Gradle plugins for Flex projects. It generates IDE configuration files and puts the dependencies from the Gradle/Maven cache on the IDE's build path. It consists of subplugins for all known Flex IDE's which can be applied separately. As of GradleFx v0.5 only the FlashBuilder subplugin has been implemented.

If you want support for all known IDE's load the plugin like this:

```
apply plugin: 'ide'
```

In any other case just apply the required subplugins.

11.1 Sub-plugins

There is a plugin for each of the following IDE's; each plugin has its matching task:

IDE load plugin execute task

IDE	Load plugin	Execute task
FDT	apply plugin: 'fdt'	gradle fdt
FlashBuilder	apply plugin: 'flashbuilder'	gradle flashbuilder
FlashDevelop	apply plugin: 'flashdevelop'	gradle flashdevelop
FlashBuilder	apply plugin: 'flexbuilder'	gradle flexbuilder
IntelliJ IDEA	apply plugin: 'ideafx'	gradle idea

The IDEA plugin was named `ideafx` to avoid conflicts with the existing 'java' `idea` plugin. All these plugins exist, however only `flashbuilder` is operational in GradleFx v0.5.

Every IDE plugin depends on the Scaffold plugin (cf. *Templates Plugin*) that generates the directory structure and the main application file.

Each of these plugins also has a matching **clean** task; for instance you could remove all the FlashBuilder configuration files from a project by executing `gradle cleanFlashbuilder`.

11.2 FlashBuilder plugin

Load the plugin:

```
apply plugin: 'flashbuilder'
```

Run the associated task:

```
gradle flashbuilder
```

With all conventions the output for a `swf` application might be something like this:

```
:my-first-app:scaffold
Creating directory structure
    src/main/actionscript
    src/main/resources
    src/test/actionscript
    src/test/resources
Creating main class
    src/main/actionscript/Main.mxml
::my-first-app:flashbuilder
Verifying project properties compatibility with FlashBuilder
    OK
Creating FlashBuilder project files
    .project
    .actionScriptProperties
    .flexProperties

BUILD SUCCESSFULL
```

To clean the project, i.e. remove all FlashBuilder configuration files:

```
gradle cleanFlashbuilder
```

CHAPTER

TWELVE

TEMPLATES PLUGIN

12.1 Overview

The Templates plugin is a feature similar to gradle-templates that can generate default directory structures and/or classes. As of GradleFx v0.5 this plugin has only very partially been implemented. Actually only the automatic generation of directory structure and the main application file (+ the descriptor file for AIR projects) is currently available, as it is a dependency required by the *IDE Plugin*. Further development is not on our priority list for the time being.

Load the plugin like so:

```
apply plugin: 'templates'
```

12.2 Sub-plugins

As of GradleFx v0.5 only one sub-plugin exists:

- Scaffold plugin: generates directory structure and main application class

This means that at the moment *apply plugin: 'templates'* and *apply plugin: 'scaffold'* will both result in the same tasks being available.

12.3 Scaffold plugin

Load the plugin:

```
apply plugin: 'scaffold'
```

The `scaffold` task is now available to you. It is the only available task for now. To use it execute `gradle scaffold` at the command line.

With all conventions this will result in the following output for a `swf` project:

```
$ gradle scaffold
:my-first-app:scaffold
Creating directory structure
        src/main/actionscript
        src/main/resources
        src/test/actionscript
        src/test/resources
```

```
Creating main class
        src/main/actionscript/Main.mxml

BUILD SUCCESSFUL
```

12.3.1 Application descriptor

In an `air` or `mobile` project an application descriptor file will also be created:

```
src/main/actionscript/Main-app.xml
```

12.3.2 Localization

If you've defined some locales in your build script (say `locales = ['nl_BE', 'fr_BE']`), directories for these locales will also be created:

```
src/main/locale/nl_BE
src/main/locale/fr_BE
```

CHAPTER

THIRTEEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*