Easy Native Extensions

Second edition

Your guide to creating an iOS ANE in under an hour

Easy Native Extensions, second edition

by Radoslava Leseva

Find us on the web at www.diadraw.com
To report errors, please send a note to office@diadraw.com

Project Editor: Hristo Lesev
Code Tester: Hristo Lesev
Cover and Interior Design: Radoslava Leseva
Proof Reader: Stephen Adams
Cover Photo: Radoslava Leseva

Notice of Rights

Notice of Liability

Trademarks

Contents

Is this book for you?

Do you…

- develop iOS mobile applications with Adobe AIR?
- have experience with ActionScript and Adobe AIR, but not so much with Objective-C and Xcode?
- want to create native extensions for iOS for your own use or for sale?
- need examples you can copy, paste and see running …
- … and at the same time understand what the code does, instead of blindly copy-pasting and spending hours tweaking?
- want to save days digging through online manuals and forums?
- prefer following diagrams and screenshots, rather than long wordy explanations?

If you answered YES to at least four of these questions, then this book is for you.

What you need

- **Mac OS X** with **Xode 6.0** and **iOS SDK 8.0 or newer;**
- **Flash Builder 4.6 or 4.7** with Adobe **AIR SDK 16.0 or newer**;
  For instructions on how to overlay the AIR SDK on the Flex SDK see Adobe's article Overlay AIR SDK on Flex SDK | Flash Builder.
- a stop watch.

I am only partly joking about the stop watch, by the way. The tutorials in this book are structured in a way that lets you whizz through them by following screenshots and not reading a word of text. Let us know how long it takes you to get an ANE and a test app up and running - drop us an email at office@diadraw.com.

Software editions used in the tutorials

The examples in this book have been tested with:

- Flash Builder 4.6 and 4.7
- AIR SDK 3.4 to 17.0
- Xcode 4.5 to 6.4, running on Mac OS X Lion, Mountain Lion and Yosemite
- iOS SDK 6.0 to 8.3

Why iOS only?

We believe in understanding the big picture, but when it comes to practicalities, we find it most effective to eat the elephant one bit at a time. If this is your first time writing a native extension for Adobe AIR, it can be overwhelming to try and understand what's going on on several platforms at once.

With that said, we are in the process of crowd funding the Android edition of this book. You can claim your copy here:
http://easynativeextensions.com/android-ebook

What our readers say

"Well all I can say is that it was easily the best $30 I've spent in a while. I was back up and running in under a day and as an added bonus Radoslava does a brilliant job of detailing how to wrap up the whole build, packaging, and deployment process into a single Ant script. Her build script was much better than the crummy one I remember cobbling together for my original ANE attempts. One of the things I hadn't really tackled previously was learning how to properly debug my ANEs. Thankfully a companion debugging guide came bundled with the package, which takes you through the steps required to write a native iOS project that will wrap around a test AIR app. With that you'll be able to add breakpoints to Xcode and inspect your ANE's native library." Read more…

Christopher Caleb
Author of Flash iOS Apps Cookbook
www.yeahbutisitflash.com

"Your ebook on iOS native extensions (at least the part of it that I've read so far) is the best, clearest communication about software development that I've ever read. It is rare and completely refreshing to find programmers who can speak just as fluently and cleverly in their 'natural' language as in code, and who can also use elegant graphics to clear-up abstractions. Bravo, and thanks!"

Craig Umanoff
Moving Pictures

"@DiaDrawCom The book was great, very good tutorial and was easy to setup my custom ane. Thanks!"

@rudyvdblom

"Your eBook on ANE's is one of the best investments I have made. The excellent explanations and example code have saved me hours of trial-and-error. And I had planned to spend a couple of weeks developing custom email and dropbox extensions. Being able to download well-documented, ready-to-build source code gives me time to add extra features to my app. Very cool. Thanks."

Andrew Rapo
Quahog Entertainment

Native extension facts

Adobe AIR native extensions are…

- a way to write and run platform-specific code in an Adobe AIR application.
- a way to provide functionality which is not accessible through the Adobe AIR SDK.
- implemented as reusable libraries.
- loaded at run-time.
- resolved (linked) at packaging time.

A native extension can help you to…

- access device capabilities not normally accessible through the Adobe AIR SDK: camera settings, gyroscope sensor, vibration, licensing payment, advertisement APIs, push notifications, etc.
- reuse existing native code.
- improve performance by allowing you to implement time or resource-critical code as close to the native platform as possible.

Disclaimer

I have purposefully created the tutorial in this book not to do anything 'useful' on the iOS side. When trying out new concepts for the first time I often find it hard to sort the wheat from the hay at the start - in this case the infrastructure of a native extension from the native side work it is meant to do. This book is about showing you the infrastructure, so you are free to add any native functionality you want.

For examples of 'useful' native extensions have a look some of those we have published.

Native extension ingredients

At the end of the chapter you will have:

1. Freedom to improvise with your own ideas, break and bend the rules by:
   - understanding of what goes into an Adobe AIR native extension for iOS;
   - knowing how the 'ingredients' of the extension fit together and communicate with one another;
   - knowing how the iOS extension fits in an application that uses it.

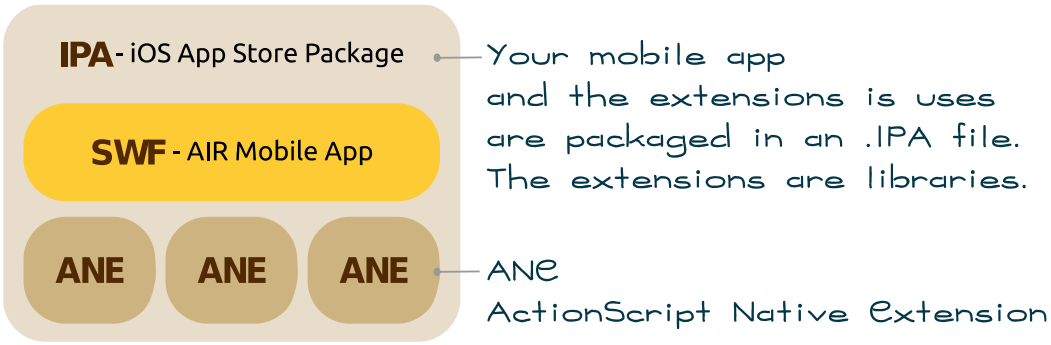
2. A convenient visual guide and reference that you can come back to at any point as you create your extension. Of course, you don't need to get to the end of the chapter to have that. :)

Tip: If you, like me, tend to learn by doing and would rather get your hands dirty first, leave this chapter for later and jump to Create your extension. All of the fundamental ideas in it are linked to relevant parts of this chapter, so just click on a link if you need more detail at any point.

What is a native extension in practice?

Let us start peeling the onion from the outside.

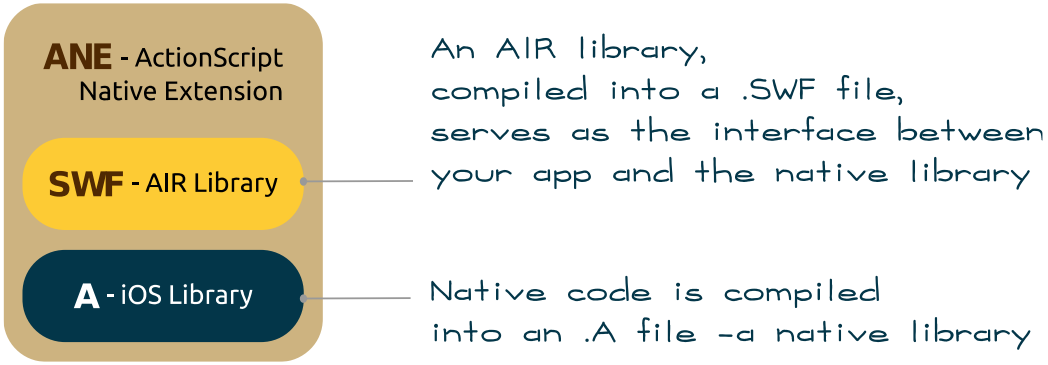
You can think of an Adobe AIR native extension as a library, packaged in an ANE file, that an AIR application can use. If you are making a mobile app for iOS with Adobe AIR, you will likely have to package the app in an IPA file - iOS App Store Package. This IPA will contain both the compiled app itself, as well as the native extensions that the app uses:



The native extension package - ANE

The native extension itself is comprised of two layers:

- **native code**, which implements platform-specific functionality – iOS-specific in this case;
- **ActionScript code**, a wrapper around the native code, which knows how to communicate with the native side and provides an ActionScript interface - an API - to be called by the application that uses the extension.

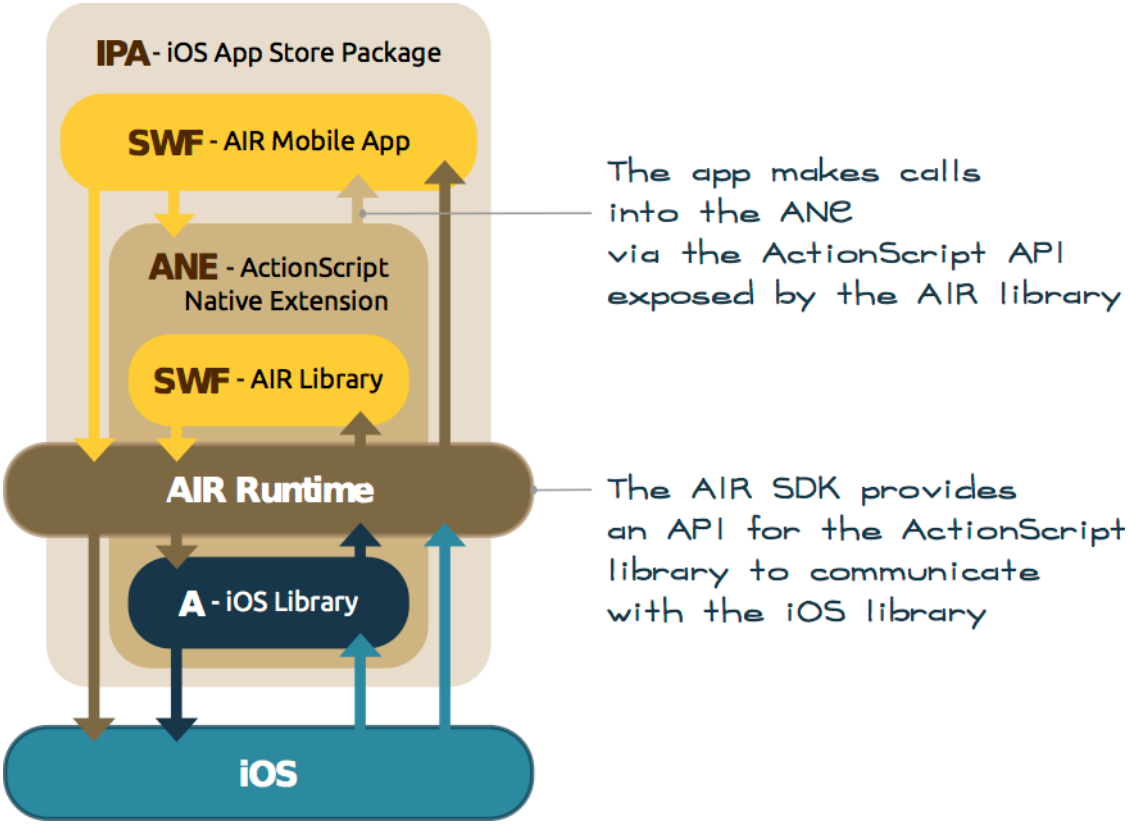


Both the native and the ActionScript layer are implemented as libraries: an iOS **.A** library and an ActionScript **.SWF** library.

Tip: An ANE file is an archive file (like a ZIP) with an .ane extension.

IPAs are from Mars, ANEs are from Venus

If part of the extension speaks the app's mother tongue, ActionScript, and another part speaks foreign (I mean native), how do app and extension communicate? Let us examine the diagram from the top down:



- An application communicates with the native extension through the extension's ActionScript layer. The app never has to speak native.

- The extension's ActionScript layer (the AIR library) is a wrapper around the native code. It doesn't have to directly speak native either. The Adobe AIR Runtime helps with that by providing an ActionScript API in the form of a class: flash.external.ExtensionContext.

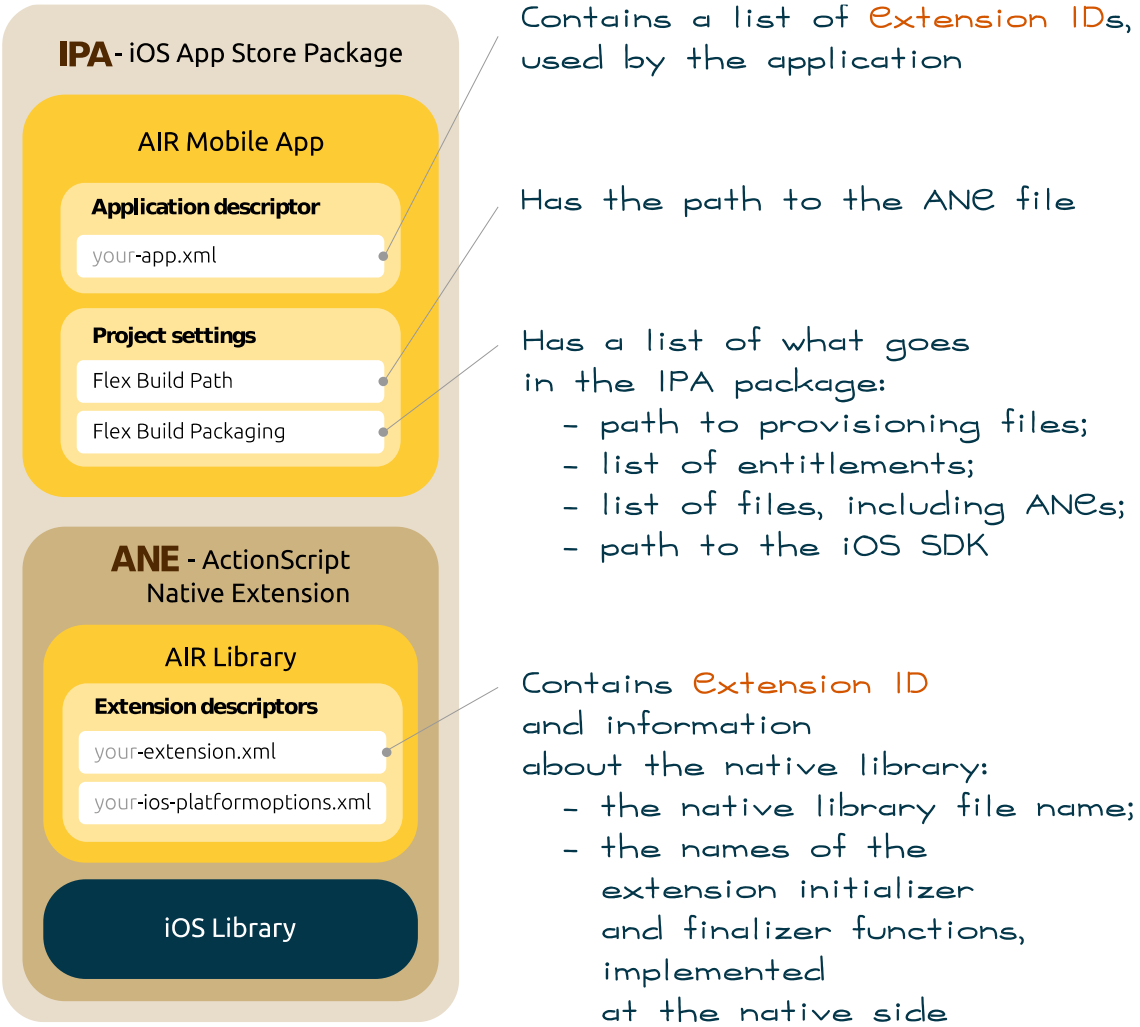
- The AIR Runtime also provides a C API which the native code must implement. The Runtime will make calls into the native code through this API.

- Finally, your native library is the layer which is in touch with the mobile platform through the platform's SDK - the iOS SDK in this case.

Note: This diagram shows the debug version of an AIR mobile app for iOS. The release version is compiled to ARM assembly code and does not require a copy of the AIR Runtime to be installed with it.

Letting the app get to know the extension

In order for the iOS library, the AIR library and the app to work together, you have to do some formal introductions. These involve setting up descriptor files for the application and for the extension and adding information to the application's project settings.



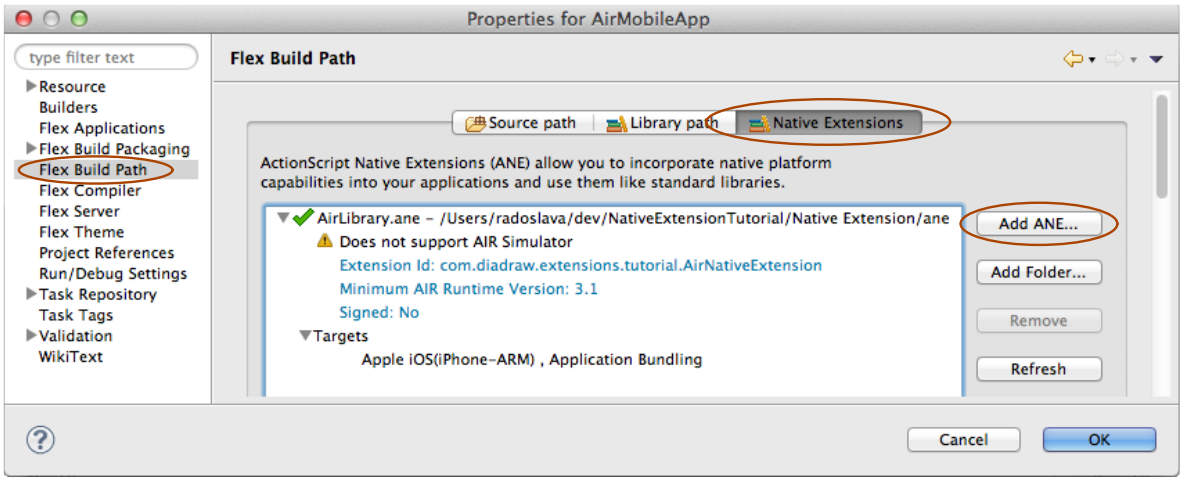
- each native extension has its ID: a unique string identifier. It is defined in he **<id></id>** entry of an XML extension descriptor file:

```
<id>com.diadraw.extensions.tutorial.AirNativeExtension</id>
```

- the IDs of the extensions that an application uses are listed in the application's descriptor - another XML file:

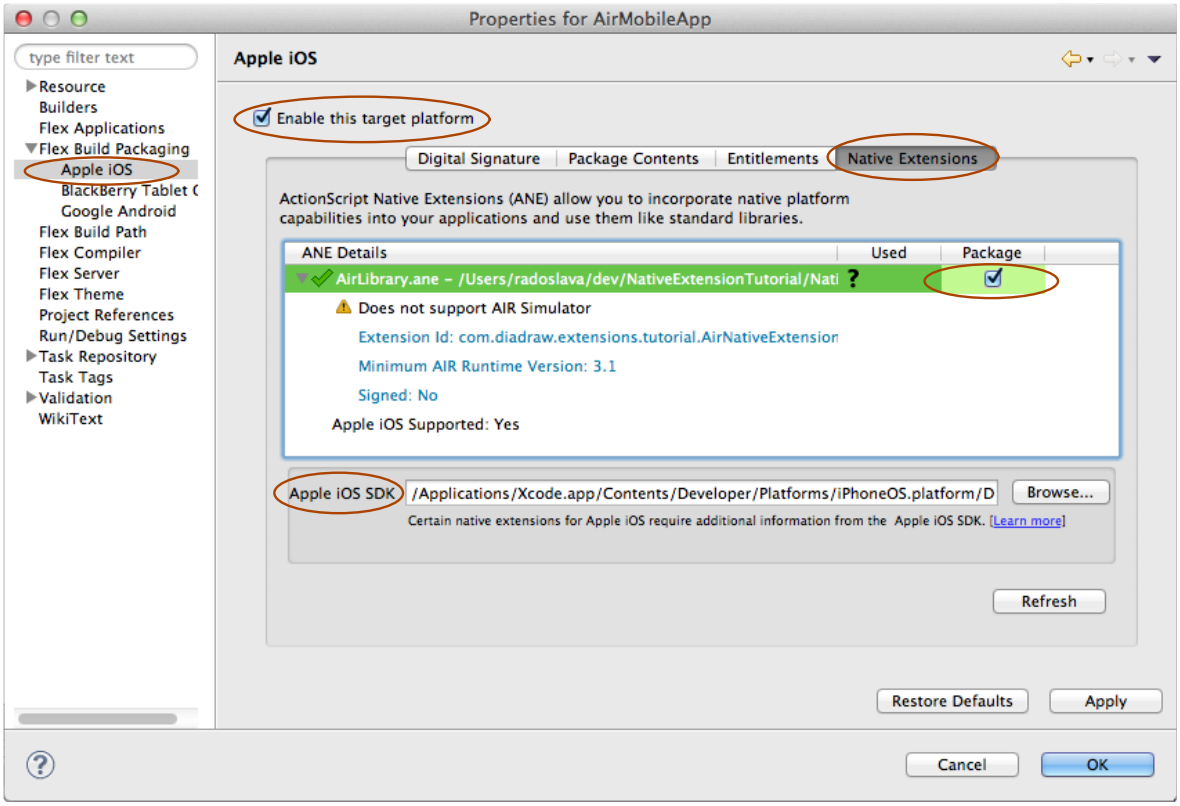
```
<extensions>
   <extensionID>com.diadraw.extensions.tutorial.AirNativeExtension</extensionID>
</extensions>
```

- for the application project to know where the native extension is, you need to add the path to the extension's ANE file to **Project** > **Properties** > **Flex Build Path**:



- to have the native extension included in the application package, you need to tick the box in the **Package** column in **Project** > **Properties** > **Flex Build Packaging** > **Apple iOS** > **Native Extensions**.

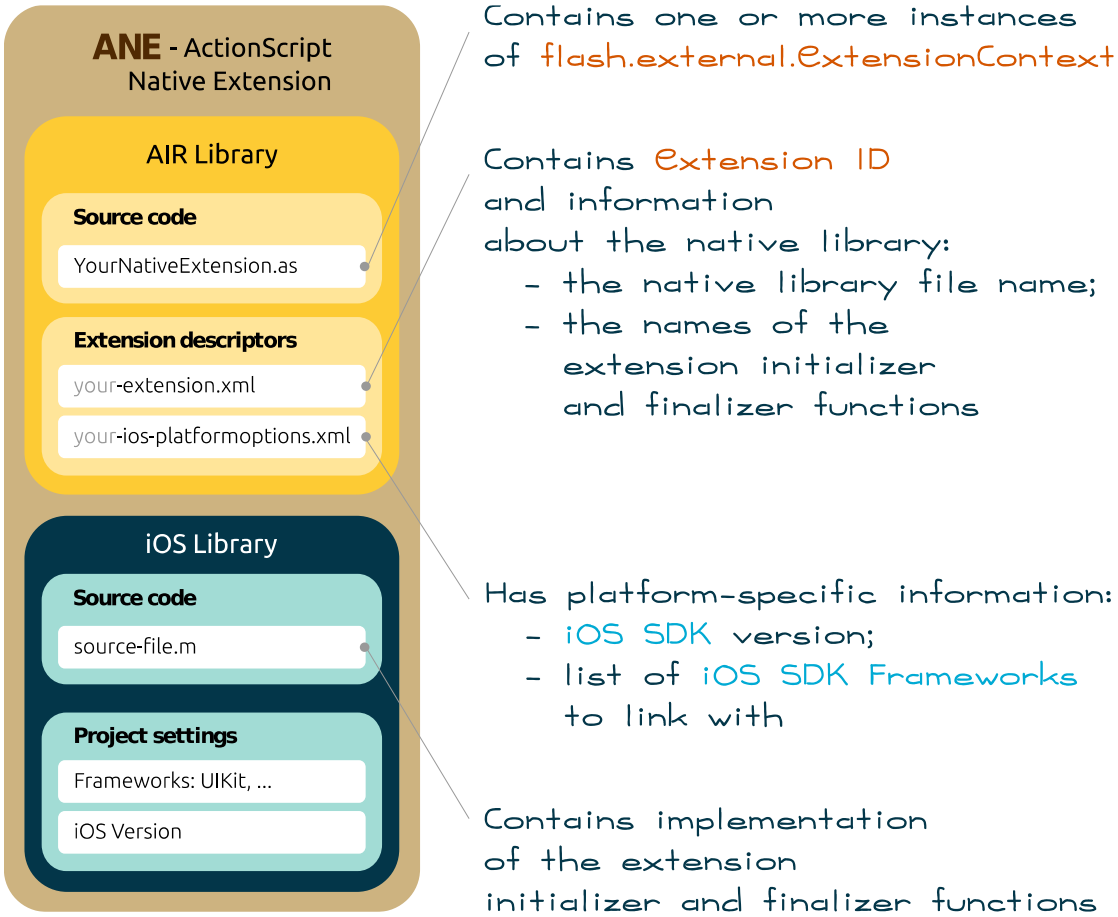
- you also set the path to the **iOS SDK** in the same dialog:



Note: The path to the iOS SDK is optional and is necessary only if any linking with native frameworks or libraries is done when packaging the app. If you are building tour app on Windows you won't even have an iOS SDK.

Introductions inside the ANE:

letting the AIR library know about the native library



Having the AIR library know about the native code it wraps involves the following:

- putting the **native library's file name** in the extension descriptor (**extension.xml**);
- adding the **extension's initializer** and **finalizer function names** to **extension.xml**;
- listing platform-specific information in the platform extension descriptor (**platformoptions.xml**). This includes the platform SDK version and any linker flags or settings (for example, a list of iOS SDK frameworks to be linked with the extension).
- implementing the extension's initializer and finalizer functions in native code and giving them the same names that you listed in **extension.xml**.

Need more detail? See The native extension descriptor files for example extension descriptors.
See diagrams of how they are used in Loading and unloading an iOS native extension.

The native extension descriptor files

For an iOS native extension you need two descriptor files: **extension.xml** and **platform.xml**. You can use your own naming convention for these, for example **your-extension.xml**, **your-ios-platformoptions.xml**.

Below are examples of both kinds of descriptor files.

extension.xml

```
<extension xmlns="http://ns.adobe.com/air/extension/16.0">
  <id>com.diadraw.extensions.tutorial.AirNativeExtension</id>
  <versionNumber>1.0.0</versionNumber>

  <platforms>

    <platform name="iPhone-ARM">
      <applicationDeployment>
        <nativeLibrary>libNativeExtensionTemplateiOS.a</nativeLibrary>
        <initializer>NativeExtensionTemplateExtensionInitializer</initializer>
        <finalizer>NativeExtensionTemplateExtensionFinalizer</finalizer>
      </applicationDeployment>
    </platform>

  </platforms>
</extension>
```

- each **<platform>** entry describes an implementation of the extension for a specific platform - iPhone-ARM in this case. If you wanted to implement an Android version of the same extension, for example, you would use the same extension descriptor file and add another **<platform>** entry under **<platforms>**.

- **<id>** contains the string identifier of your extension. The Adobe Air Runtime will use this to load the extension.

Tip: You can put any string as your extension ID, but make sure it is unique and does not accidentally match another extension's ID. Adobe recommends using a reverse DNS for the ID. For example: **com.yourCompanyName.yourExtensionName**.

Tip: The same extension ID string will have to be used in three places:
1) the **<id>** entry of the extension descriptor file;
2) the **<extensionID>** entry in the application descriptor in any app that uses the extension;
3) as an argument to the ActionScript call to **ExtensionContext.CreateExtensionContext()**, which you will implement in Step 1: Adobe AIR Library of the Create your extension chapter.

- **<versionNumber>** is the version of your extension;
- **<nativeLibrary>** contains the file name of the native library you will create in Step 2: Native Library;
- **<initializer>** and **<finalizer>** contain the names of the two functions, which the Air Runtime will call when loading (initializing) and unloading (finalizing) the extension. You will implement these in your native library project in Step 2: Native Library.

platform.xml

```
<platform xmlns="http://ns.adobe.com/air/extension/16.0">

  <sdkVersion>8.1</sdkVersion>

  <linkerOptions>
      <!-- This will link your extension with MessageUI.framework: -->
      <option>-framework MessageUI</option>

      <!-- This flag suppresses linker warnings: -->
      <option>-w</option>
  </linkerOptions>

</platform>
```

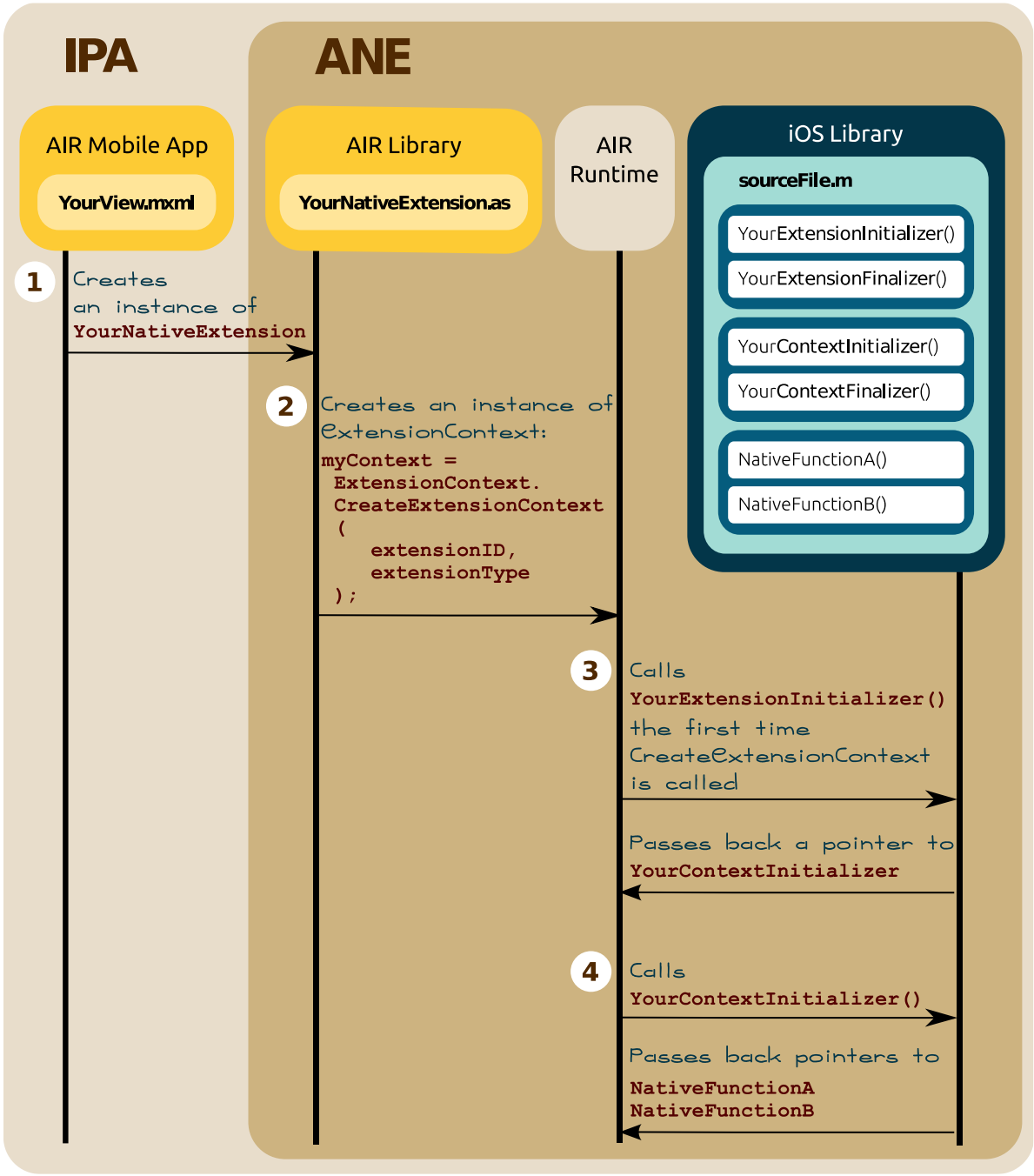
- **<sdkVersion>** refers to the version of the platform's (iOS) SDK you will compile your native library with;
- **<linkerOptions>** allows you to pass information to the linker in each **<option>** entry. This can include setting linker flags or passing a list of iOS frameworks that AIR should link to. The example above tells the linker to suppress warnings and to link with **MessageUI.framework**.

Tip: Adobe AIR links with certain iOS SDK frameworks by default, so, even if your native code uses them, you don't need to list them in **<linkerOptions>**. At the time of this writing these include: CoreFoundation, Foundation, UIKit, CoreGraphics, MobileCoreServices, SystemConfiguration, AudioToolbox, CFNetwork, QuartzCore, CoreLocation, CoreMedia, CoreVideo, AVFoundation, OpenGLES, Security.

Loading and unloading an iOS native extension

1. Loading the extension

A native extension that you app uses only gets loaded in memory if the app makes a call into it. Below is a diagram that illustrates the order of what happens:



1. The app makes a call into the native extension library for the first time, typically by instantiating the class that provides the extension's ActionScript API;

2. The ActionScript part of the extension (the AIR library) requests an instance of **flash.external.ExtensionContext** from the Air Runtime by calling

```
ExtensionContext.CreateExtensionContext( extensionID, extensionType );
```

where:

- **extensionID** is the same string identifier you defined in your extension descriptor;
- **extensionType** is a string, which you can use to define different 'types' of your extension. This is not a type in the programmatic sense, but rather a name that the ActionScript and the native side agree on. Based on what name is passed the extension can initialize its context in different ways, for example by exposing different sets of native functions and thus providing different behavior. You can pass NULL if your extension does not vary its behavior.

Need more detail?

Have a look at Extension context for details on what a context is and how it works.

3. The AIR Runtime, in turn, calls the extension initializer function, which you have implemented in native code. Its name is listed in your extension descriptor.

The initializer function must implement the following signature:

```
typedef void (*FREInitializer)
(
        void**                  extDataToSet,
        FREContextInitializer*  ctxInitializerToSet,
        FREContextFinalizer*    contextFinalizerToSet
);
```

- **extDataToSet** points to extension-specific data, which you can optionally create and set. It will then be passed back to native code when the extension context is initialized. If you don't need such data, you can pass NULL.
- **ctxInitializerToSet** is a function pointer to an extension context initializer function;
- **ctxFinalizerToSet** is a function pointer to an extension context finalizer function.

All of these are output arguments: they are set inside the extension initializer function and passed back to AIR .

Here is what the initializer function from the diagram might look like:

```
void YourExtensionInitializer(
    void**                  extDataToSet,
    FREContextInitializer*  ctxInitializerToSet,
    FREContextFinalizer*    ctxFinalizerToSet )
{
    extDataToSet = NULL;
    *ctxInitializerToSet = &YourContextInitializer;
    *ctxFinalizerToSet = &YourContextFinalizer;
}
```

Tip: In your implementation you can name your extension initializer and finalizer functions whatever you want. Make sure however that you give them **unique** names. All extensions that an app uses are loaded into the same namespace and having two initializer functions with the same names will cause only one of the extensions to be loaded.

4. Now that the Runtime has a pointer to the extension context initializer, it makes a call into it. The role of the extension context initializer is to let the Runtime know what native functions it can call. Its signature looks like this:

```
typedef void (*FREContextInitializer)
(
        void*                   extData,
        const uint8_t*          ctxType,
        FREContext              ctx,
        uint32_t*               numFunctionsToSet,
        const FRENamedFunction** functionsToSet
);
```

- **extData** points to the extension-specific data, which you created in the extension initializer function or is set to NULL if you do not have use for such data.
- **ctxType** represents the type of the extension context you want to create. Using context types is optional. They are defined by you and allow you to vary the behavior of your extension.
- **ctx** points to the instance of **flash.external.ExtensionContext** which you use at the ActionScript side. At the native side it is represented by the **FREContext** type, which is in essence a **void***.
- **numFunctionsToSet** is an output argument, which you need to set to the number of native functions your extension exposes to the AIR Runtime;
- **functionsToSet** is another output argument, which tells the AIR Runtime what the pointers to the exposed functions are. For that you need to set **fucntionsToSet** to an array of **FRENamedFunction** objects.

Let us have a look at what the context initializer from the diagram would look like.

Example implementation of a context initializer:

```
void YourContextInitializer(
      void*                         extData,
      const uint8_t*                ctxType,
      FREContext                    ctx,
      uint32_t*                     numFunctionsToSet,
      const FRENamedFunction**      functionsToSet )
{
  static FRENamedFunction extensionFunctions[] =
  {
       { (const uint8_t*) "as_nativeFunctionA", NULL, &NativeFunctionA },
       { (const uint8_t*) "as_nativeFunctionB", NULL, &NativeFunctionB },
  };

  *numFunctionsToSet = sizeof(extensionFunctions) /
                       sizeof(FRENamedFunction);

  *functionsToSet = extensionFunctions;
}
```

Note how information about functions is passed: each function is represented by a structure like this:

```
typedef struct FRENamedFunction_
{     const uint8_t* name;
      void*          functionData;
      FREFunction    function;
} FRENamedFunction;
```

where:

- **name** is a string by which the function will be known to the extension context;
- **functionData** points to a lump of data which will be passed as an argument every time the function is called from ActionScript. This is optional and you can set functionData to NULL.
- **function** is a pointer to the function.

So, why set the name of NativeFunctionA as 'as_nativeFunctionA'?

The answer is: does not matter what string you put in the function name string, as long as you use the exact same string to call the function from ActionScript.

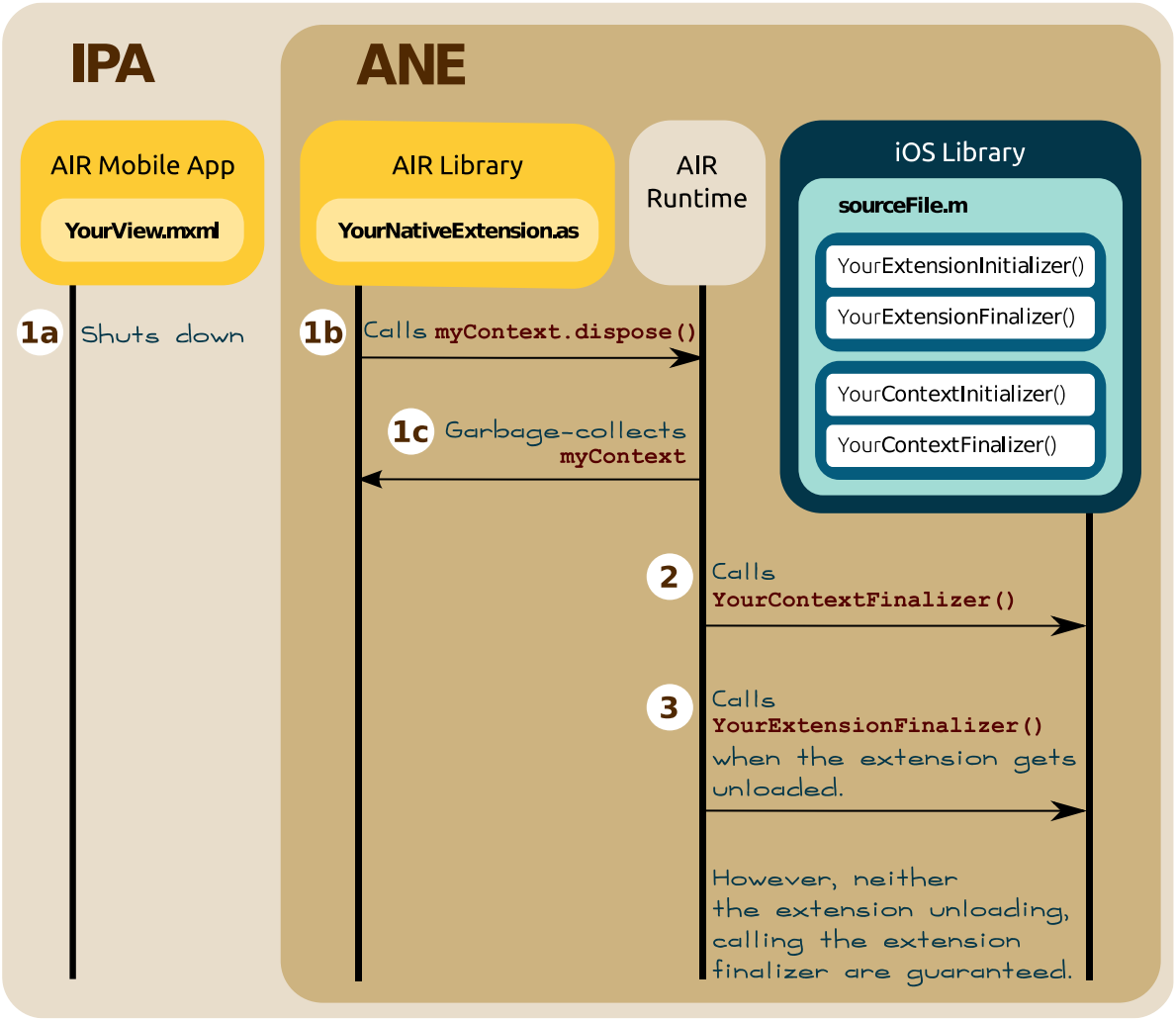
Want more? If you are curious how this comes into play when functions are called from ActionScript, jump to Calling native functions from ActionScript.

With an extension context in place and knowing which your native functions are and what they are called you are good to go. Initialization done.

2. Unloading the extension

The extension unloading follows a similar scenario to extension loading, except:

- unloading can be triggered by **one** of three different things - have a look at **1a**, **1b** and **1c** below;
- there is **no guarantee that an extension will be unloaded** at all.



To cause an extension to be unloaded, **one of three things** (but not all three) needs to happen:

1a. The app shuts down and is unloaded from memory.

1b. Your AIR Libary calls

```
myContext.dispose();
```

1c. The AIR Runtime garbage collector detects no references to the context instance and disposes of it.

2. This causes the Runtime to call the context finalizer function. The role of this function is to allow you to clean up any context-specific resources you might have allocated. It has the following signature:

```
typedef void (*FREContextFinalizer)( FREContext ctx, );
```

3. Then the Runtime calls the extension finalizer function, where any global extension-specific data can be cleaned up. Its signature looks like this:

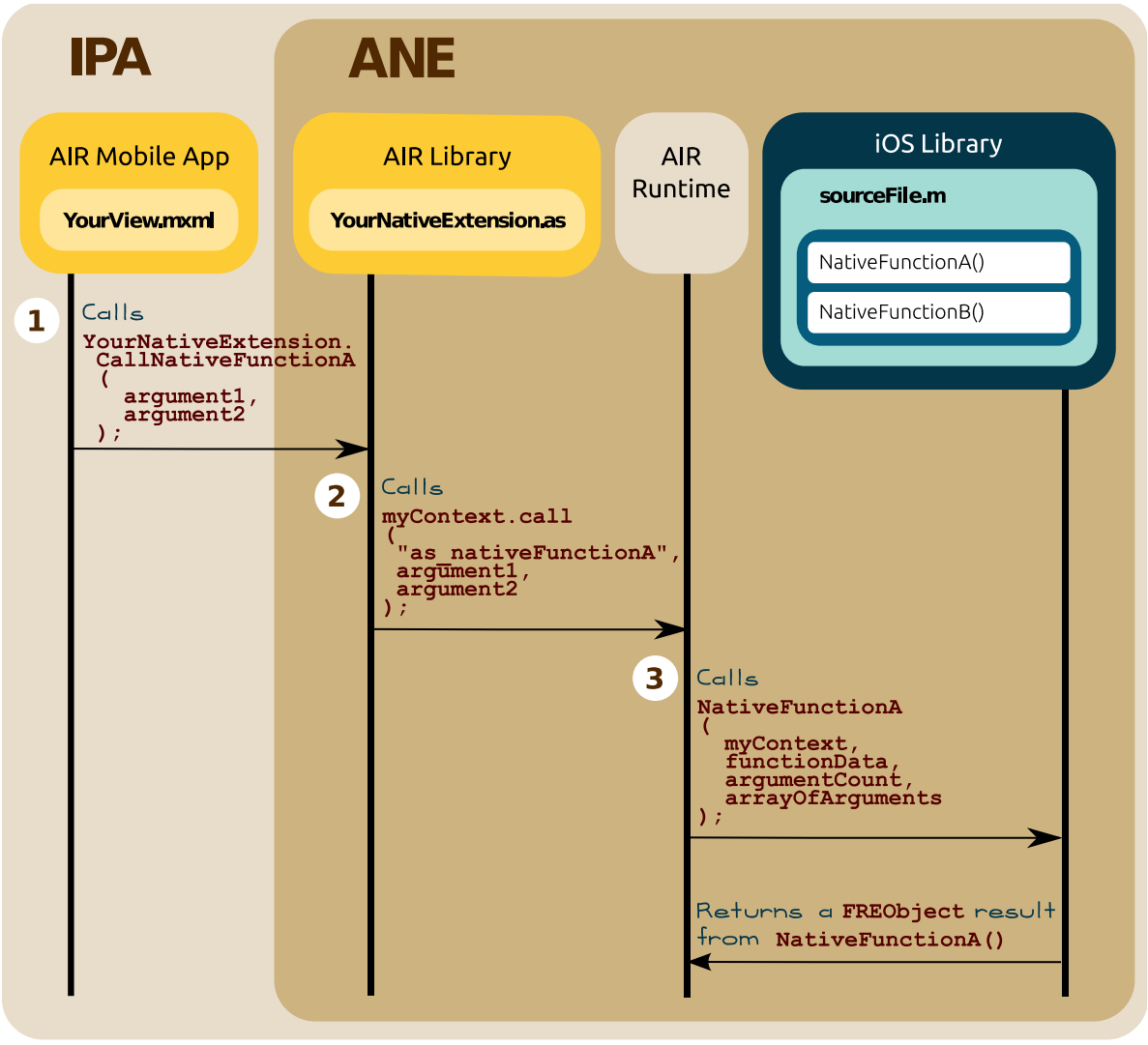
```
typedef void (*FREFinalizer)( void* extData );
```

Calling native functions from ActionScript

After formal introductions have been made between the various bits of code:

- ActionScript in the app;
- ActionScript in the AIR library;
- and native code (usually Objective-C) in the native library,

your app can start making calls to the native extension. Here is the full story:



1. The app calls one of the AIR library's APIs, let us name it **CallNativeFunctionA** and say that it takes two arguments:

```
nativeExtInstance.CallNativeFunctionA( argument1, argument2 );
```

where nativeExtInstance is an instance of YourNativeExtension.

2. Inside **CallNativeFunctionA()** the AIR library makes a call to

```
myContext.call( "as_nativeFunctionA", argument1, argument2 );
```

where **as_nativeFunctionA** is the name of the function which the extension context has mapped to the function pointer of **NativeFunctionA()**. If you are a die-hard C or C++ coder like me and like utilizing the compiler as much as possible, you are likely to cringe at this way of making calls. Ce'st la vie.

Tip: If you wonder why this example is passing a strange string which doesn't necessarily match the name of NativeFunctionA() and how the extension context knows what to call, flip to the context initializer example - it is the initializer's fault. Me innocent.

3. On the native side this call is translated to:

```
NativeFunctionA( myContext, functionData, argumentCount, arrayOfArguments );
```

where:

- **myContext** is a pointer to the extension context you created in your AIR library. Note that on the native side it arrives as type **FREContext**, which is just **void***.
- **functionData** is a pointer to data, which you set (or did not set) in your context initializer to be passed to the function every time it is called;
- **argumentCount** and **arrayOfArguments** are how **argument1** and **argument2** get passed to native code: pointers to them are put in an array and the array and its size are passed to NativeFunctionA. For those of you with pedantic inclinations[1]: the array actually contains **FREObject** instances, which are again nothing more than **void***.

Tip: At this point you are probably wondering how different types of data are passed between ActionScript and native code and how they are treated. Well, I have created a couple of examples for you. In return all you have to do is get your hands dirty. Deal?
If you are up for it, roll up your sleeves and go to the Create your extension chapter.

[1] Email me!

For a very long list of types and how to convert them back and forth between ActionScript and native code we have a special book, called iOS vs. ActionScript Data Types Guide. It helps you build a library of one-line conversion functions for most types of data you will need to deal with.

Any native function you want to expose to ActionScript has to implement the following signature:

```
typedef FREObject (*FREFunction)
(
    FREContext ctx,
    void*      functionData,
    uint32_t   argc,
    FREObject  argv[]
);
```

Look familiar? If not, have a look at point **3** above or on the diagram.

Note that the function must return a **FREObject** as a result. This is one of the ways to pass data back to your AIR Library. If your native function doesn't need to return a result, return NULL.

Tip: For two more ways to get data from the native side to the ActionScript side of your extension have a look at the next section, Three ways to pass data from native code to ActionScript.

Three ways to pass data from native code to ActionScript

Breathe. This will be a short section.

When you make a call from ActionScript to a native function, it has three ways to communicate back to the ActionScript side of your extension. Two of them are synchronous, one is asynchronous.

Returning a function result

Each Objective-C function you expose to ActionScript has the same signature:

```
FREObject ASGetStringLength(
      FREContext ctx,
      void* funcData,
      uint32_t argc,
      FREObject argv[] );
```

The **FREObject** the function returns as a result is one way of passing data back to ActionScript. The **FREObject** type is just **void ***. The AIR C API has convenience functions that wrap basic and custom types as **FREObject**s for you, for example **FRENewObjectFromInt32**():

```
FREObject ASGetStringLength(
      FREContext ctx,
      void* funcData,
      uint32_t argc,
      FREObject argv[] )
{
      int32_t value = /* get the length of the incoming string */

      FREObject intValue = NULL;
      FRENewObjectFromInt32( value, &intValue );

      return intValue;
}
```

At the ActionScript side the result arrives as type **Object**, which you can cast to the type you expect:

```
public function getArgumentLength( _argument : String ) : int
{
      return m_extensionContext.call( "as_getStringLength", _argument ) as int;
}
```

For a full list of the **FRENewObject*** functions see Functions you use in Adobe's online manual.

Using output parameters

An output parameter is a parameter passed to a function, which you can modify, so that the function's caller gets the modified value.

In the example below an ActionScript **Array** is passed to the native function that reverses the order of the elements in it:

ActionScript

```
public function reverseArrayInPlace( _array : Array ) : void
{
      m_extensionContext.call( "as_reverseArrayInPlace", _array );
}
```

Objective-C

```
FREObject ASReverseArrayInPlace(
   FREContext ctx, void* funcData, uint32_t argc, FREObject argv[] )
{
    FREObject array = argv[ 0 ];
    assert( NULL != array );

    uint32_t arrayLength = 0;
    FREResult status = FREGetArrayLength( array, &arrayLength );
    assert( FRE_OK == status );

    int32_t lastElementIndex = arrayLength - 1;
    int32_t endOfFirstHalf = arrayLength / 2;

    for ( int32_t i = 0; i < endOfFirstHalf; ++i )
    {
        swapArrayElements( array, i, lastElementIndex - i );
    }

    return NULL;
}
```

Parameters are passed to native code in an argument array of **FREObject** instances. The AIR Native C API has convenience functions that allow you to extract basic and custom type values from these **FREObject**s. For example, **FREGetObjectAsInt32()**. For a full list of the **FREGetObjectAs*** functions see Functions you use in Adobe's online manual.

Sending an asynchronous event to ActionScript

Sending asynchronous events offers native code a way of communicating what is going on at moments when returning a function result or using an output argument are not an option. For example, native code can emit an event to let you know that a certain window finished initializing or that an error was thrown while it was connecting to an online service.

The AIR Native C API offers you a method, called FREDispatchStatusEventAsync[2], which causes an event to be emitted that can be caught by the ActionScript code. The event type you need to listen for in ActionScript is of type flash.events.StatusEvent.

The code in this example will emit a **flash.events.StatusEvent** that will be caught by ActionScript:

Objective-C

```
FREObject ASSendMeAMessage(
    FREContext ctx, void* funcData, uint32_t argc, FREObject argv[] )
{
    FREDispatchStatusEventAsync(
        ctx,
        ( const uint8_t * ) "This is the message CODE",
        ( const uint8_t * ) "This is the message LEVEL" );

    return NULL;
}
```

ActionScript

```
m_extensionContext.addEventListener( StatusEvent.STATUS, onStatusEvent );
...
private function onStatusEvent( _event : StatusEvent ) : void
{
      trace( _event.code );
      trace( _event.level );
}
```

Tip: FREDispatchStatusEventAsync() requires a pointer to the ExtensionContext (**FREContext**). All functions that ActionScript can call receive a **FREContext** as a function argument. To be able to call **FREDispatchStatusEventAsynch()**, make a copy of the **FREContext** that is passed to the context initializer function.

For full examples of passing data between Objective-C and ActionScript see the tutorial in the Create your extension chapter.

[2] **FREDispatchStatusEventAsync** is the only call you are allowed to make from any thread. All other calls into the AIR C API need to be made on the main thread.

Extension context

Think of the extension context as the middleman between ActionScript and native code. It is the kind tour guide that speaks both languages, helps both sides communicate with one another and keeps the good tone of the conversation[3].

On the ActionScript side

The extension context is of type flash.external.ExtensionContext and lets ActionScript code:

- call native functions:

```
m_extensionContext.call( "as_reverseArrayInPlace", _array );
```

- receive events, sent from Objective-C:

```
m_extensionContext.addEventListener( StatusEvent.STATUS, onStatusEvent );
```

- create one or multiple instances of the extension context:

```
m_extensionContext = ExtensionContext.createExtensionContext(EXTENSION_ID, null);
```

On the Objective-C side

The extension context masquerades as **FREContext**, which knows the mapping between native functions and ActionScript and you can use it to emit asynchronous events.

[3] As in:
Native Code: "What a ridiculous hat!"
ActionScript: "What did he say?"
Context: "He thinks your eyes are adorable, madam…"

Summary

- An Adobe AIR native extension is implemented as a library.
- An extension contains native code, which makes calls to the native platform (iOS) and ActionScript code, which acts as a wrapper for the native code.
- An application which uses a native extension only makes calls into the ActionScript wrapper layer of the extension.
- The ActionScript and the native side of a native extension communicate through the extension context.
- The extension context is set up when the native extension is loaded and initialized.
- ActionScript can pass data to native code in the parameters to the native functions it calls.
- Native code can pass data back to ActionScript in three different ways:
    - as a result of a function call;
    - as an output parameter;
    - by sending an asynchronous event.

Create your extension in under an hour

At the end of the chapter you will have:

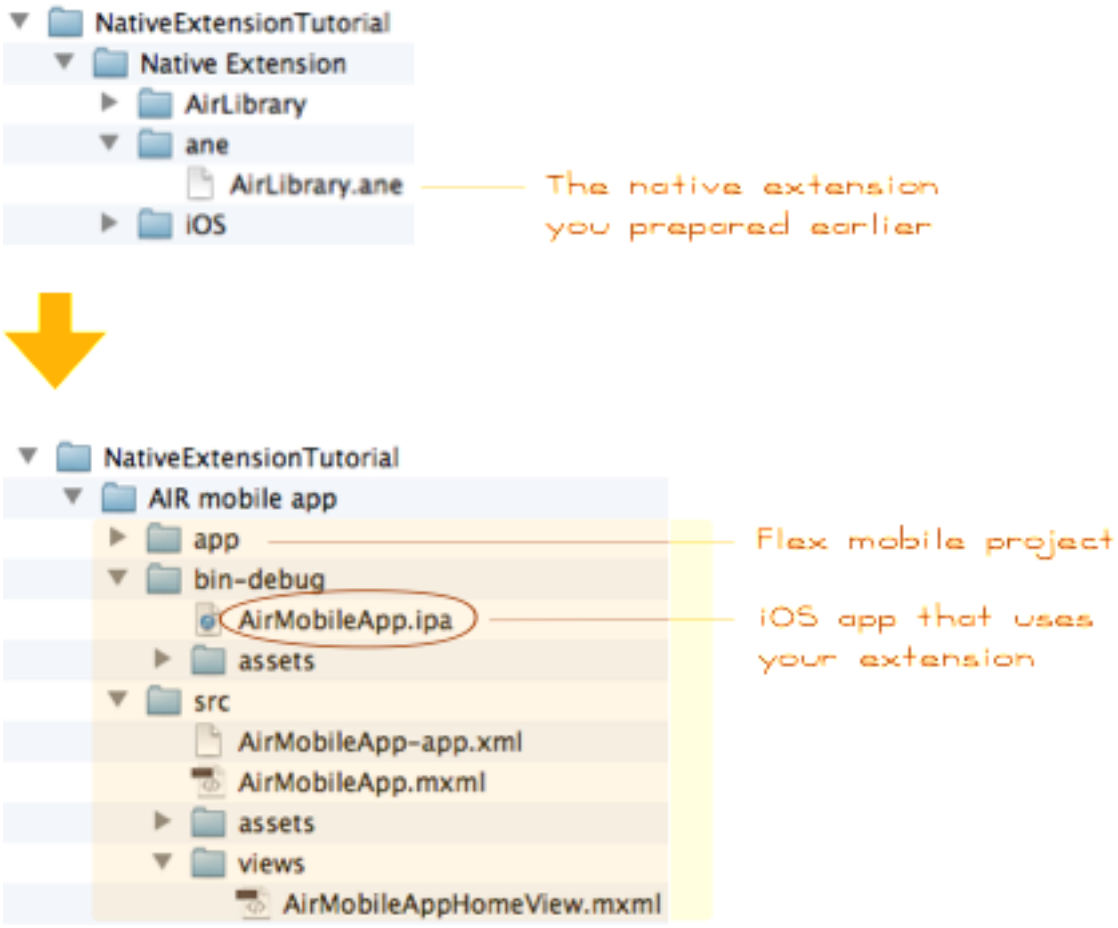
Is that stopwatch handy?

Time to start it in 3, 2, 1...

Step 1: Adobe AIR Library, 15 minutes

At the end of this step you will have:

- A Flash Builder project in your **AirLibrary** folder with the following structure:



Tip: For the purpose of this tutorial, it would be easiest if you follow the structure shown in the examples when you set up your projects. However, by no way do you have to do that to achieve a working extension. If you are following this tutorial while making your own extension, feel free to use a structure and naming convention that work best for you and make changes where necessary.

1.1. Create folders for your projects

1.2. Set up an AIR Library project in Flash Builder

1.3. Create the extension descriptors

1.4. Create your extension class

1.5. Add the APIs to the extension class

API 1: Send me a message

API 2: Take a string argument, return a number

API 3: What type is my argument?

API 4: Reverse array in place

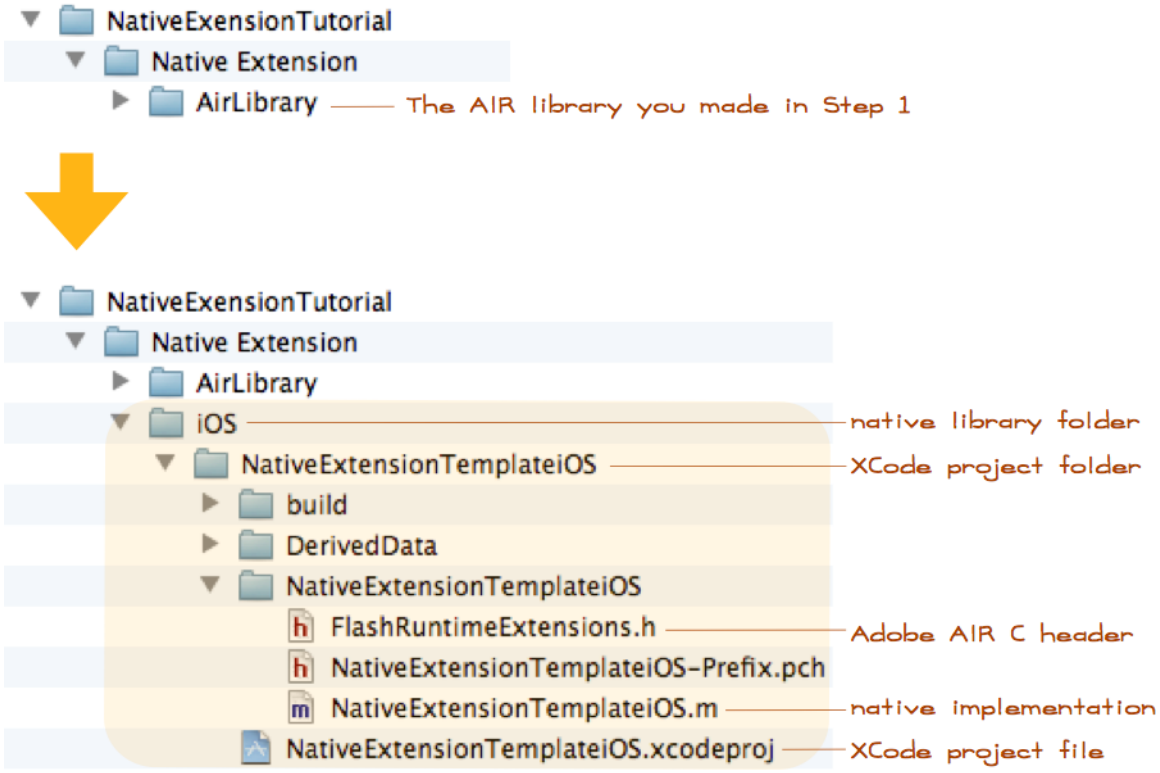
1.5. Add an event handler for messages from the native side

1.6. Build your AIR Library

Step 2: Native Library, 15 minutes

At the end of this step you will have:

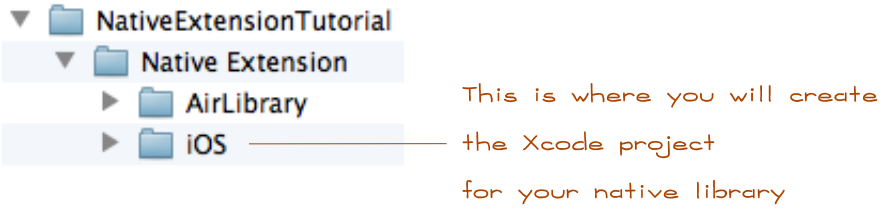
- An Xcode project in your **iOS** folder with the following structure:



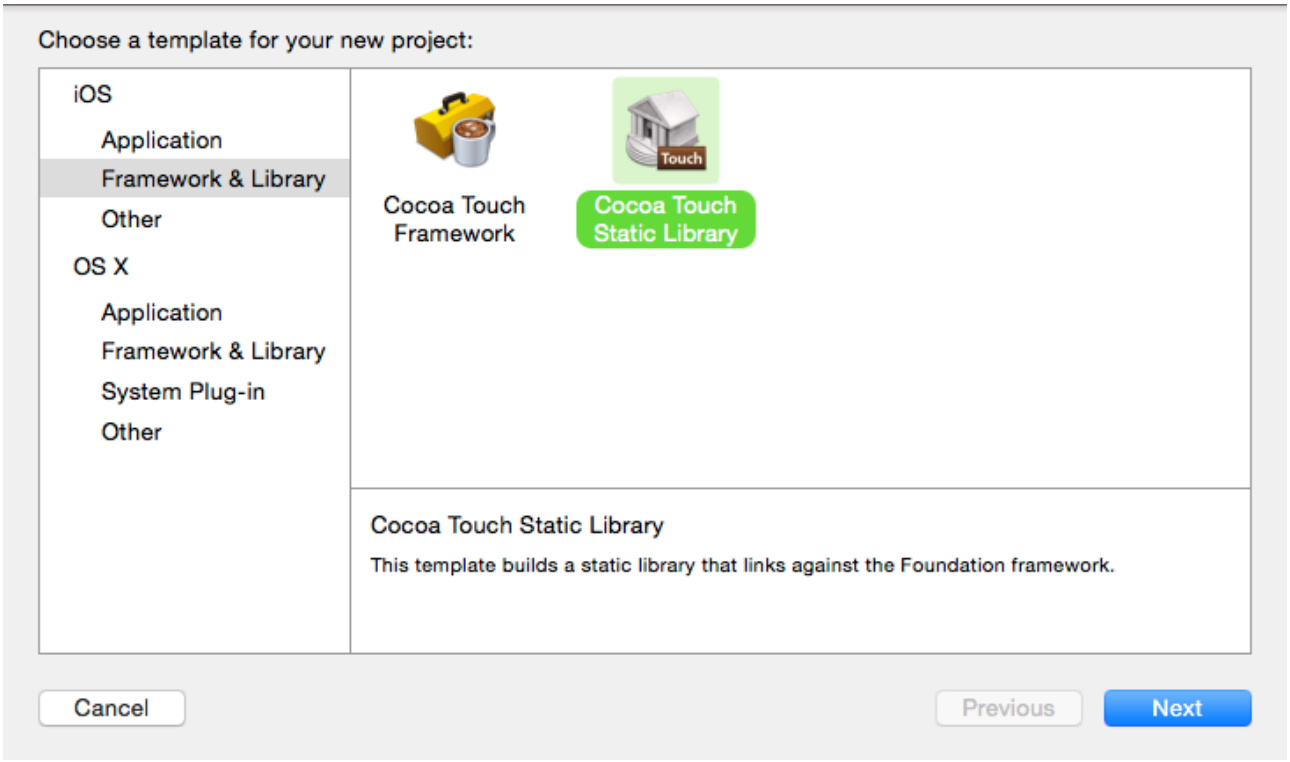
Tip: As mentioned in Step 1: AIR Library, if you would like to structure your project folders differently, feel free to do so. You will have to reflect that structure in the build scripts at the end, but it will not be a major change, I promise.

2.1. Set up an Xcode project

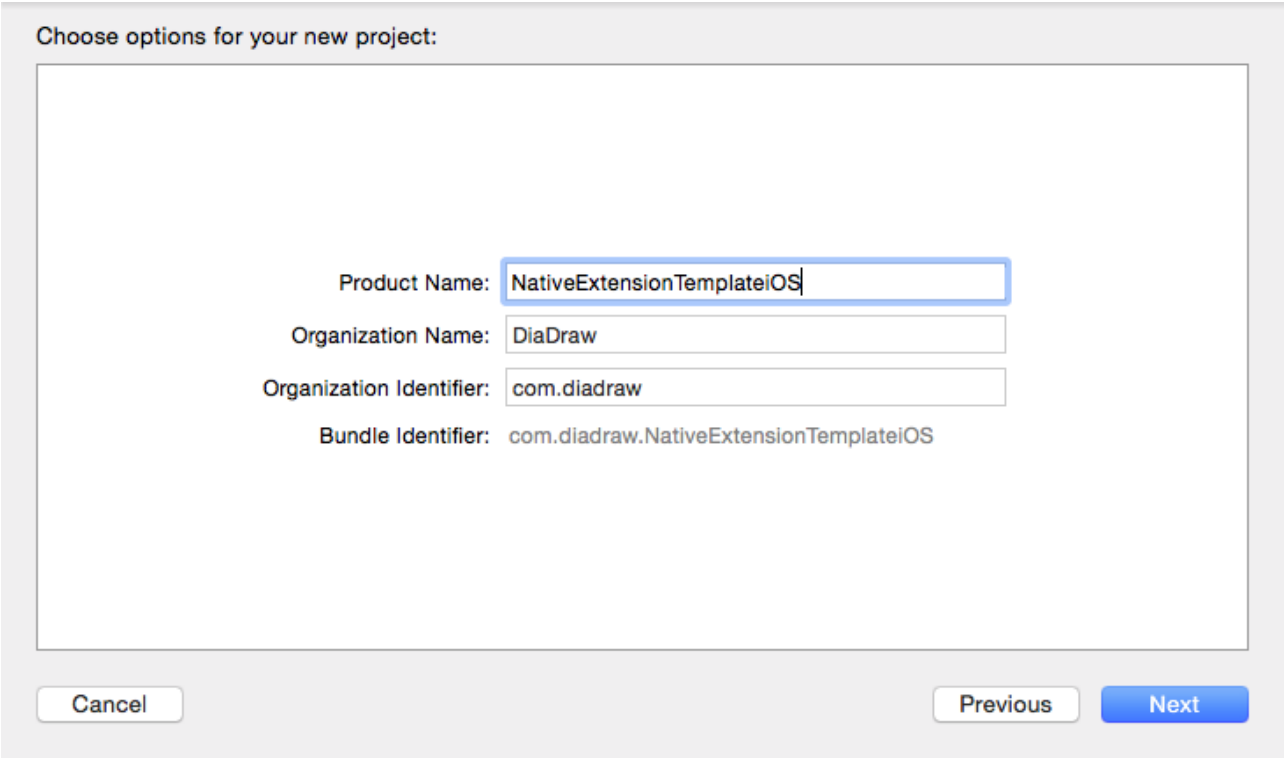
- Make a subfolder in the **NativeExtensionTutorial/Native Extension** folder and name it **iOS**. This is where you will create the Xcode project for your native library.



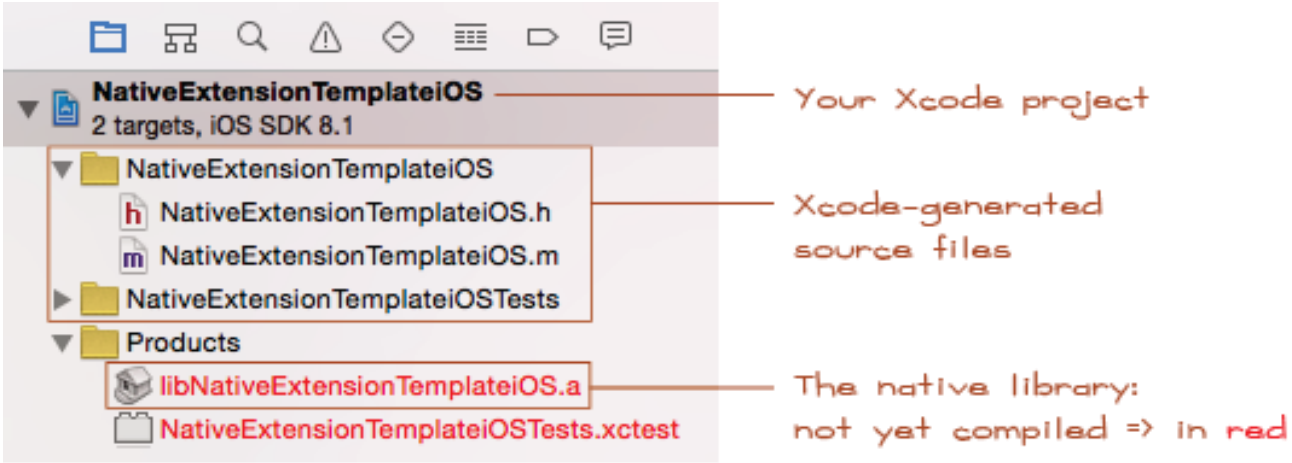
- Start Xcode and from the main menu select **File** > **New** > **Project**. In the new project dialog, under **iOS** on the left, select **Framework & Library** and on the right select **Cocoa Touch Static Library**:



- Name your project **NativeExtensionTemplateiOS** and save it in the **NativeExtensionTutorial/Native Extension/iOS** folder you made earlier:



This is what your project should look like in Xcode's **Project Navigator**:



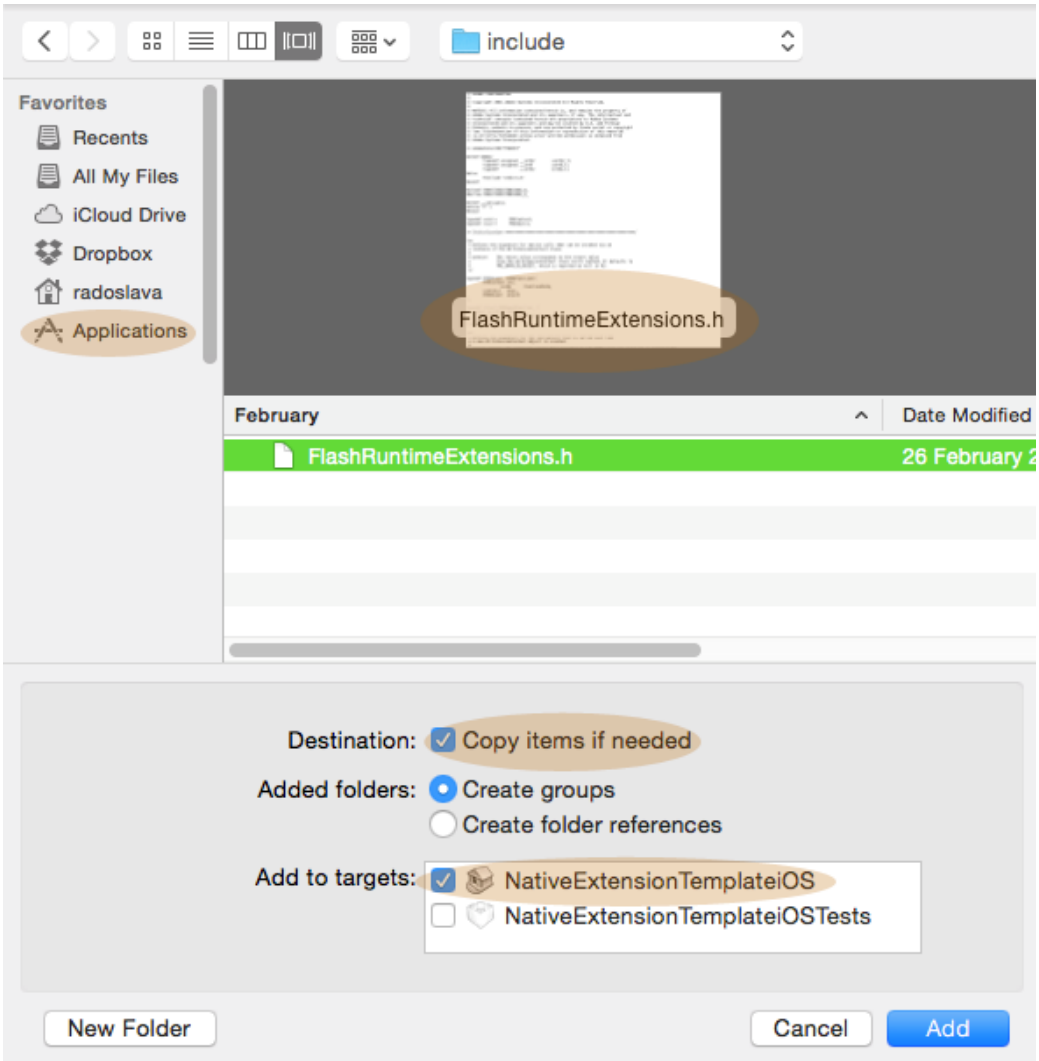
Tip: You will not necessarily need the header file that Xcode has generated for you, so feel free to get rid of it: right-click **NativeExtensionTemplateiOS.h** and select **Delete**.

2.2. Add FlashRuntimeExtensions.h

FlashRuntimeExtensions.h should be in your **AIR SDK/include** folder. This header file defines a C API with functions and structures, which can be accessed from ActionScript via the **flash.external.ExtensionContext** class. In your native library you will use some of these structures and will provide implementations for functions from the C API.

```
/Applications/Adobe Flash Builder 4.6/sdks/4.6.0/include/FlashRuntimeExtensions.h
```

Select **File > Add Files to ...** and navigate to **FlashRuntimeExtensions.h** to add it to your project.



Tip: The choice is yours whether you allow **FlashRuntimeExtensions.h** to be copied into your project folder. For the pros and cons of either choice see this article: [Do you need a copy of FlashRuntimeExtensions.h in your project?]

2.3. Get coding: implement the extension APIs

If you started following this tutorial from Step 1: AIR Library (I promise not to judge you if you jumped here first), you know what these functions are and have implemented their ActionScript wrappers. Now you will see and implement what they actually do.

Tip: All native functions that you want to be able to call from ActionScript need to implement a very specific signature, as you will see below. For details on that and how they are called from ActionScript have a look at Calling native functions from ActionScript.

Import FlashRuntimeExtensions.h

First things first. You will be using the API defined in this header, so import it. Open **NativeExtensionTemplateiOS.m** and add this at the top:

```
#import "FlashRuntimeExtensions.h"
```

Import Foundation.h

Foundation.h is part of the Apple iOS API and defines primitive types and classes for Objective-C. You will need it in order to use the NSString class, for example.

```
#import <Foundation/Foundation.h>
```

Implement the functions that your AIR Library will call

Native function 1: Send me a message

Native function 2: What's the length of a piece of string?

Native function 3: What type is my argument?

Native function 4: Reverse an array in place

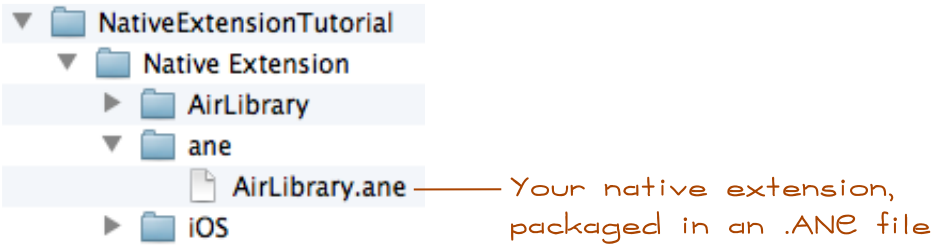
Implement the Context Initializer and Finalizer

Implement the Extension Initializer and Finalizer

2.4. Build your native library

Step 3: Package the extension, 10 minutes

At the end of this step you will have:



At the time of this writing (Flash Builder 4.6 and 4.7, AIR SDK 17.0) the Flash Builder IDE does not give you a way of packaging a native extension into an .ANE file. The AIR SDK however comes with a command-line tool, which you can use instead: the AIR Developer Tool (ADT).

The good news is that there is a way of getting Flash Builder to cooperate and let you build and package your extension with the click of a button. If you would rather cut to the chase and see how this is done, flip to Making your life easier: single-click build.

If however you would rather first get your hands dirty on the command line and see what it takes to make an ANE package, read on.

Recipe for packaging an ANE on the command line

Ingredients:

- 1 well-built AIR library: **AirLibrary.swc**
- 1 **library.swf** file, extracted from AirLibrary.swc
- 1 extension descriptor : **AirLibrary-extension.xml**
- 1 platform options descriptor : **AirLibrary-ios-platformoptions.xml**
- 1 well-built iOS library: **libNativeExtensionTemplateiOS.a**

Preparation time:

10 minutes

Serves:

As many as you dare serve your ANE to.

Preparation method:

Like this sample?

Get the full book and accompanying code here.

How did you do?

Time to check that stopwatch.

Want to do better?

Of course you do. All this copying, renaming and extracting of files and then running command-line stuff can be a nuisance, especially if you have to do it every time you want to test a small change in your code.

As you might have guessed by the cheeky way I asked the question above (if that didn't do it, the eye-rolling at 'nuisance' was probably a giveaway), I have a trick up my sleeve.

You can get all of the above work done in a single step and… drum roll… from Flash Builder. You will have to do a little bit of work to convince Flash Builder to cooperate, but trust me, it is worth each and every one of the ten minutes you will spend doing it.

Curious now? Flip to Making your life easier to see how you can have your cake and eat it.

Step 4: Test the extension in an app, 15 minutes

At the end of this step you will have:

Set up a Flex Mobile Project for your app

Set up the user interface

Add calls to your native extension

Add the extension to your project

Flex Build Path

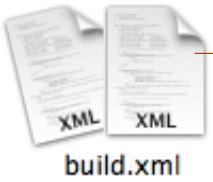
Flex Build Packaging

Build and run your app

Play >:-)

Making your life easier: single-click build

At the end of this chapter you will have:



A couple of scripts to let you build, package and deploy your extension, plus a test app, with a single click

… plus hours of saved time in the long run, as you will never again have to spend it manually copying and extracting files, creating and deleting folders, figuring out and running command line stuff, installing packages through iTunes and all of that after every small change in code you need to test.

At the click of a button in Flash Builder you will be able to:

- build your Xcode project;
- build your Air library;
- package the extension in an ANE file;
- build your test app;
- package the test app with the ANE in an IPA file;
- install the test app on your iPhone or iPad;
- do all of the above for your other projects with minimal changes in the script setups.

Many roads lead to Rome

When it comes to automating builds you have various options: shell scripts, Ant scripts, batch files… What you choose as your solution depends on your end goal.

My goal here is to minimize the time and effort between making a change either in ActionScript or in native code and having the extension and its test app built and installed on your device.

Using Adobe AIR tools to install your app on an iOS device became possible in AIR SDK 3.4. Flash Builder allows you to integrate **Ant scripts** in your project and have them run when you click Run or Debug, so Ant seems the way to go.

There are a couple of phases in automating your builds:

Phase 1: Automating the building and packaging of your native extension.

Phase 2: Automating the building, packaging and deploying of any application that uses one or more of your extensions. Alternatively, if you are using **Flash Builder 4.7**, getting the app build to trigger a rebuild of the ANE while you are still working on the ANE code.

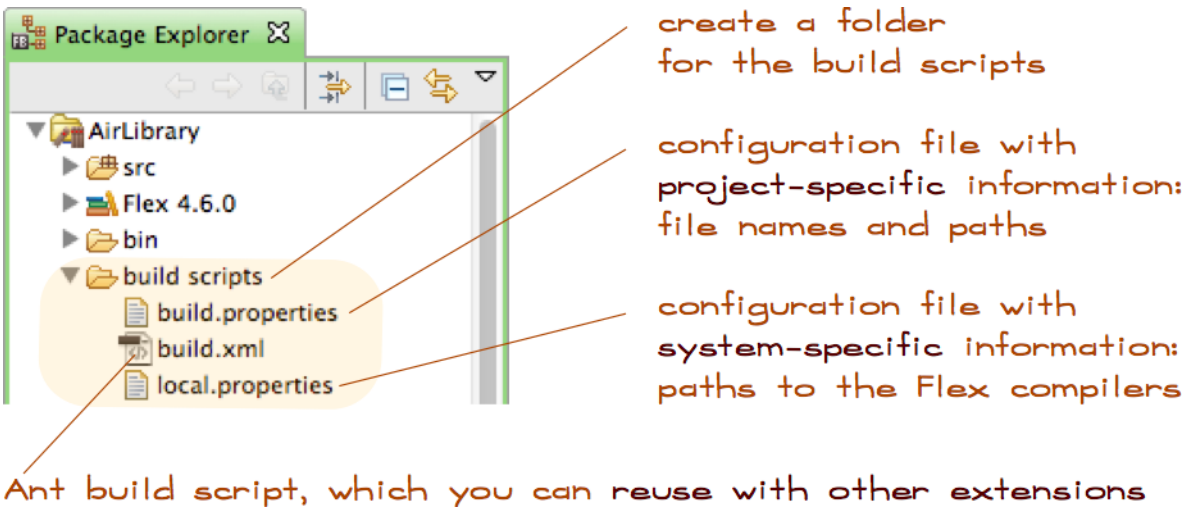
Phase 3: Integrating the automation in Flash Builder.

The scripts I am about to offer you have been used and tested in various AIR projects that use native extensions and should require minimal customization to get to work in your own projects. They are not a cure for all pains, of course, although they should give you a good starting point.

Tip: If you are new to Ant, have a look at the bonus chapter 7 Things you need to know about Ant Scripts.

Phase 1: Automate the building and packaging of your ANE

In this phase you will create an Ant build script, which you can reuse with other native extensions. You will also configure your Flash Builder project to use it, so you can build and package your ANE from within Flash Builder, rather than having to do it on the command line.



- Start by creating a folder for the script and its configuration files: with your **AirLibrary** project selected in Flash Builder's Package Explorer select **File** > **New** > **Folder**. Call the new folder **build scripts**.

- Select the folder you just created, click **File** > **New** > **File** and create a file, called **build.xml**. This will be your build script.

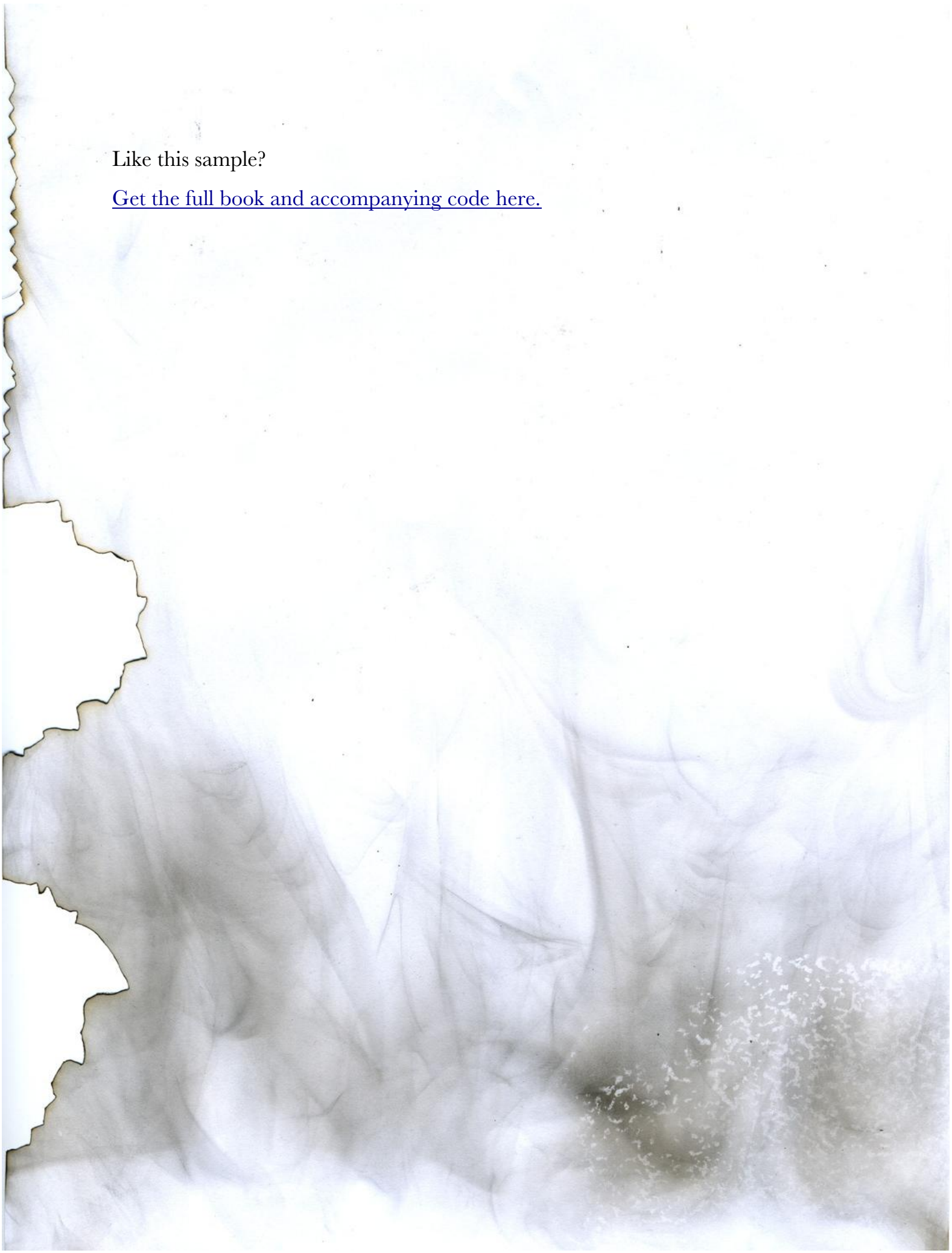
- Create another file in the same folder and call it **local.properties**. This file will contain the paths to the Flex compilers and tools.

- Finally, create another file in the same folder and name it **build.properties**. It will contain project-specific information: the Air library project name, the path to your Xcode project, the name of the ANE, etc.

Tip: **build.properties** is the only file that you will need to modify if you decide to reuse the build script with a different native extension.

Like this sample?

Get the full book and accompanying code here.



Test the build script on the command line

Try a few variations of the script

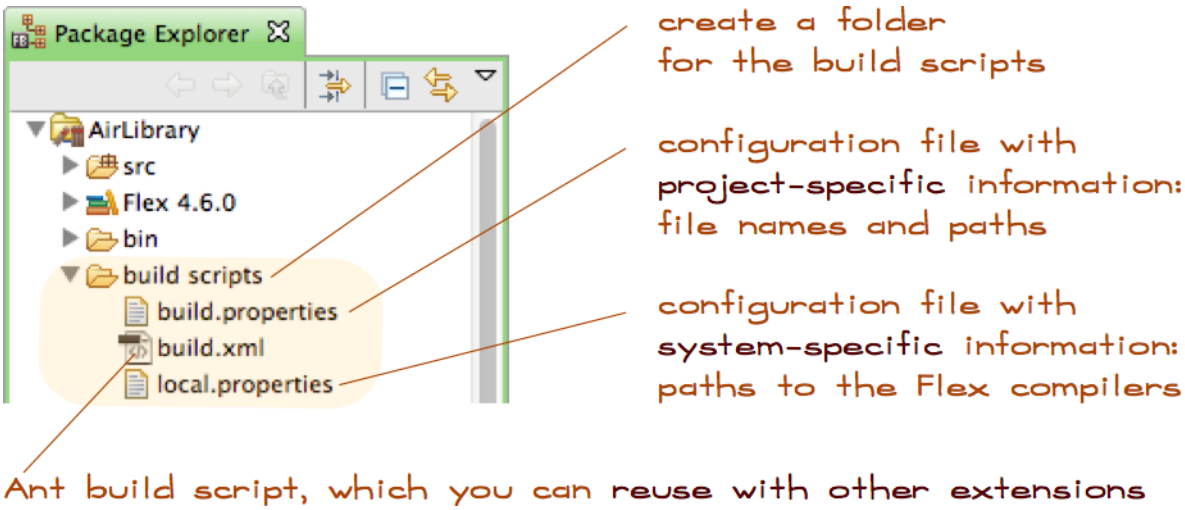
Test the build script in Flash Builder

Like this sample?

Get the full book and accompanying code here.

Phase 2: Automate the building, packaging and deployment of your test app

This will follow the exact same steps you have just taken to automate the building and packaging of the ANE. The build script will be a bit different. Let us dig into it.



- Create a folder in your **AirMobileApp** project and call it **build scripts**.
- Create the following files in the build scripts folder: **build.xml**, **local.properties** and **build.properties**. Have a quick look back at Phase one, if you want to remind yourself what the purpose of each of the files is.
- Open your app descriptor file: this should be an XML file in your **src** folder, called **AirMobileApp-app.xml** and locate the **<content>** tag inside the **<initialWindow>** tag:

```
<initialWindow>
      <!-- The main SWF or HTML file of the application. Required. -->
      <!-- Note: In Flash Builder, the SWF reference is set automatically. -->
      <content>[This value will be overwritten by Flash Builder in the output
app.xml]</content>
```

Replace the text in the square brackets with your app's SWF file name:

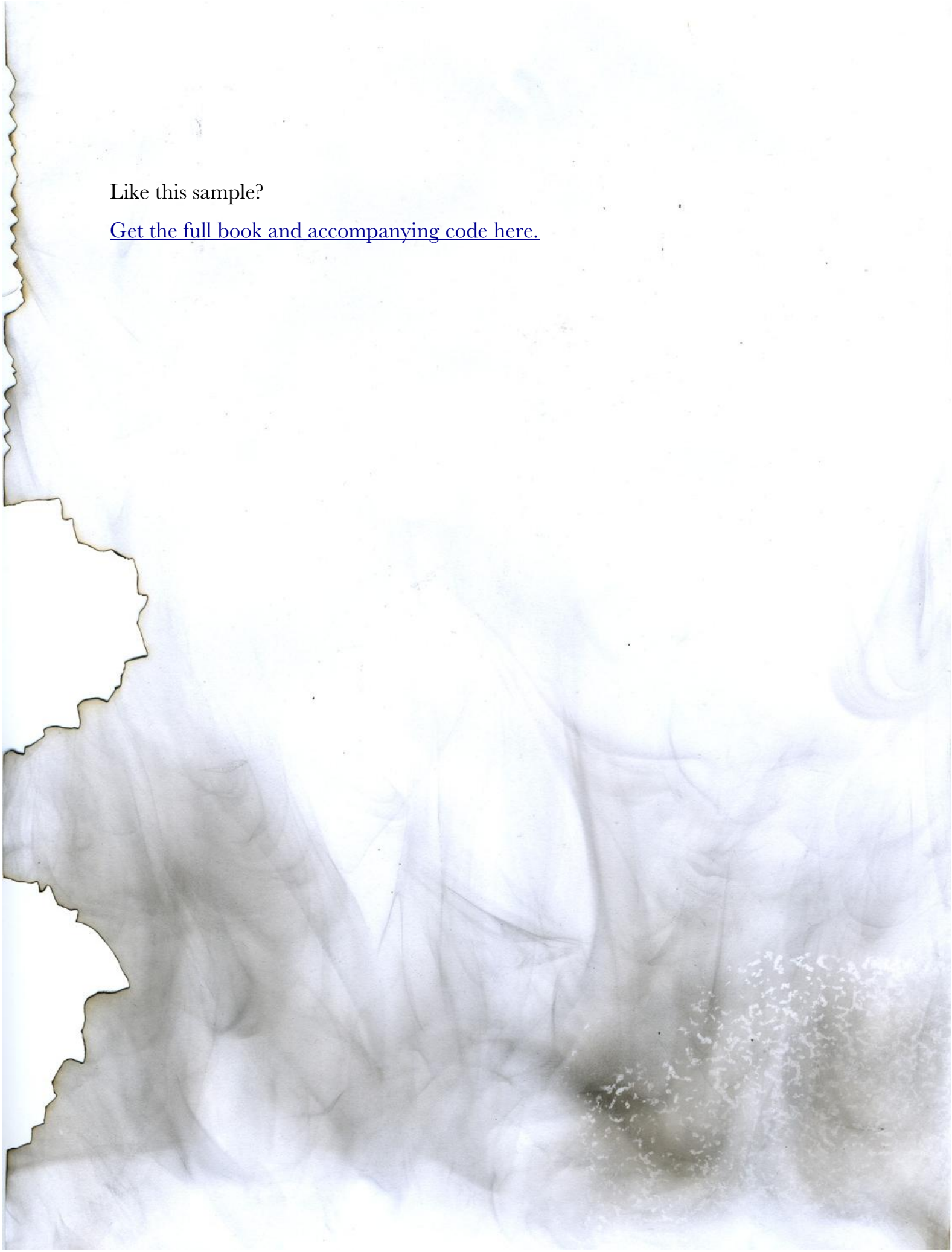
```
<content>AirMobileApp.swf</content>
```

When you build your app in Flash Builder this value is set automatically for you. However, unless you set this tag by hand, you will get this error message when you run the build script:

```
error 105: application.initialWindow.content contains an invalid value
```

Like this sample?

Get the full book and accompanying code here.



Like this sample?

Get the full book and accompanying code here.

Phase 3: Configure your project for one-click build in Flash Builder

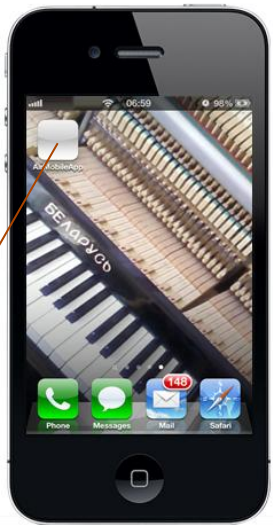
Like this sample?

Get the full book and accompanying code here.

- This should run the build script you added. Open the **Flash Builder Console** to see the script's progress (**Window** > **Show View** > **Console**):

- When the build is run, Flash Builder will tell you that the app is now ready for installing and running: just like earlier, when you did the building and installing of your app without a script, only this time you don't have to do all these steps that the dialog tells you to do: the app should now be on your device!

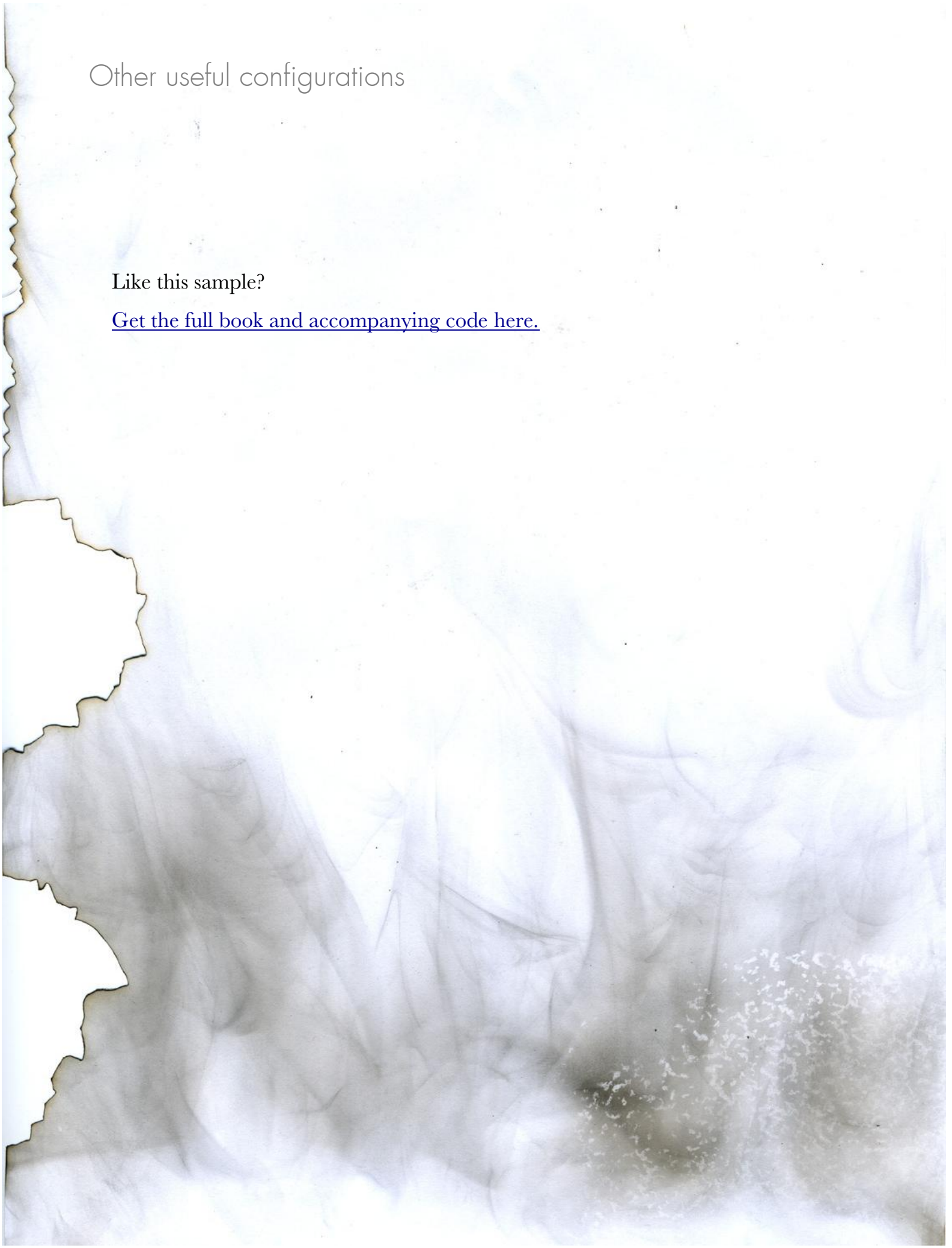
No need to follow the installation instructions this time.
The app should already be installed on your device by the script.



Tip: If you don't see the **Packaging Completed** dialog, there are probably build or packaging errors. Select **Project** > **Build Project** and check the **Flash Builder Console** to see what the errors are.

Like this sample?

Get the full book and accompanying code here.



Bonus chapter:

Adding support for the AIR Simulator

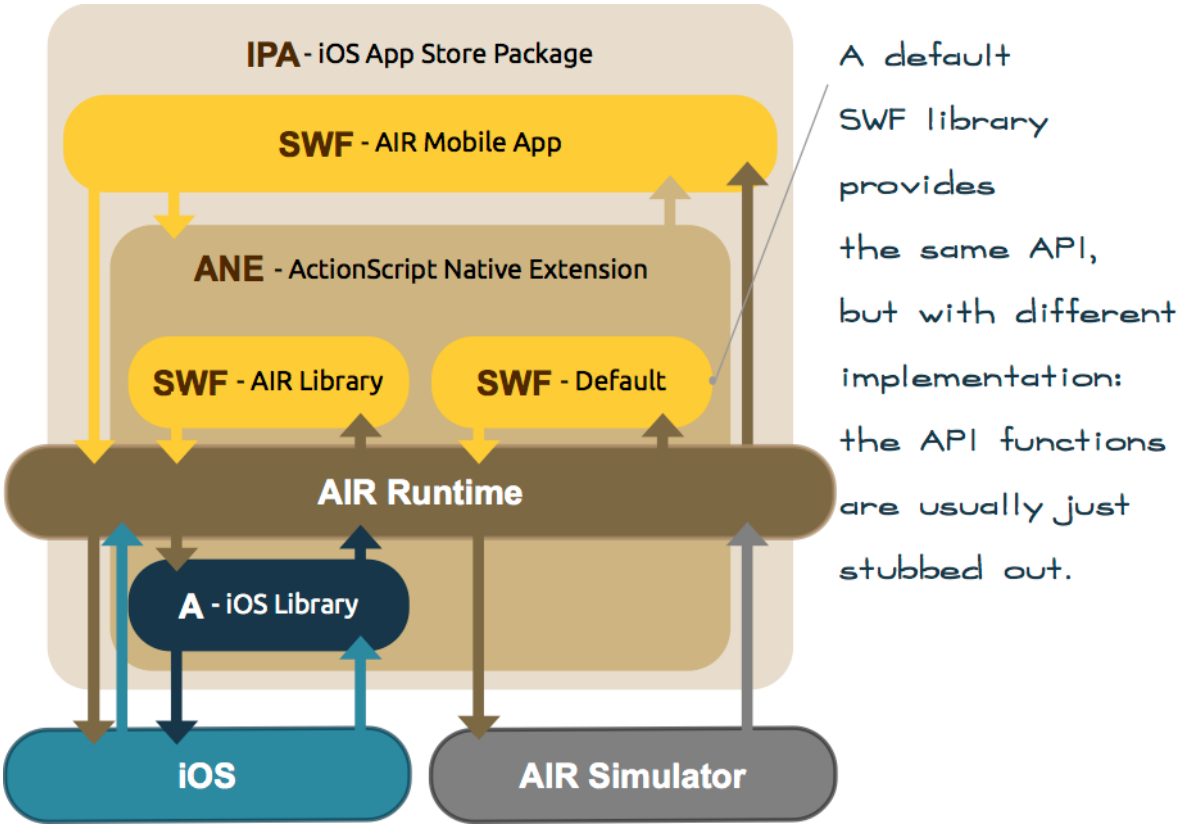
Why?

By definition your native extension is meant to run in the native environment on the device you code it for. The functionality you want to implement in it will probably not be available anywhere else: you won't be able to test how your app interacts with the gyroscope in a simulator, for example.

A native extension, on the other hand, is only a subset of the full functionality of a mobile app and the users of your extension will most likely want to be able to run and test their app in the AIR Simulator without having to install and run it on a device every time.

How?

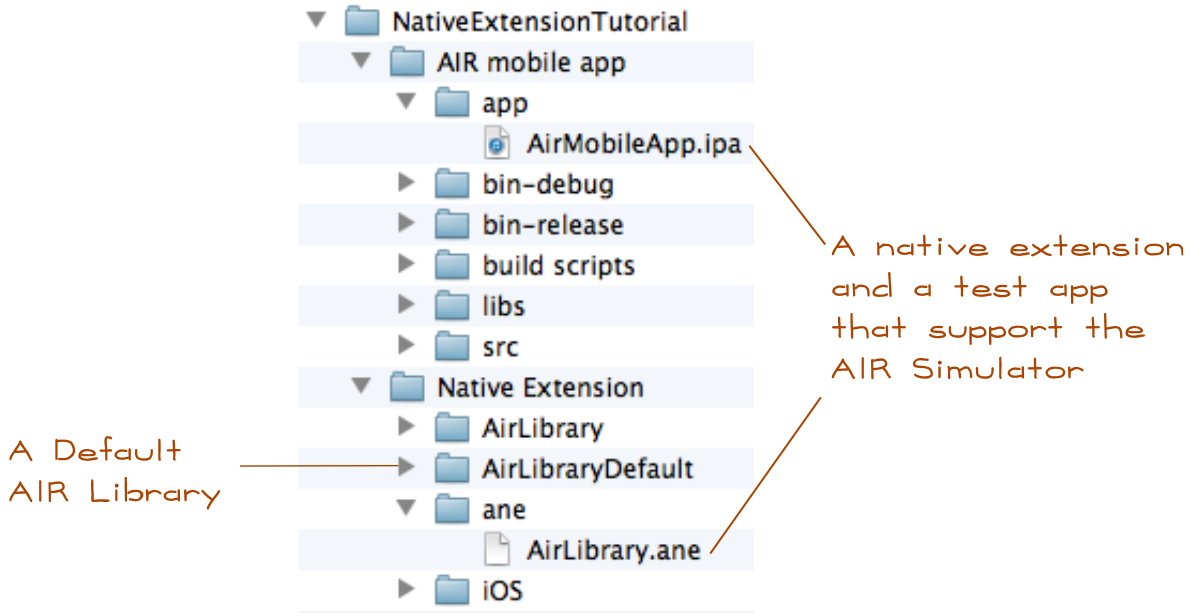
Having your native extension support the AIR Simulator requires that you provide a **Default Implementation** of your Air library. Think of it as having two different implementations of the same API.



In effect you will add another AIR library to your ANE that will expose the same methods, which will be implemented as stubs – functions that don't do much if anything.

At the end of this step you will have:

- A default implementation of your Air library
- A native extension that supports the AIR Simulator
- A package of your test app that you can run in the AIR Simulator
- An addition to your build scripts that will let you switch between building an ANE for iOS or for the AIR Simulator.



Setup your default implementation project

Package the ANE with AIR Simulator support

Run your test app in the AIR Simulator

Automate the building and packaging for the AIR Simulator

Bonus chapter:

Is your ANE 64-bit?

At the end of 2014 Apple announced their new requirements for submitting apps to the app store:

- from **February 1st 2015** all new apps must include 64-bit support;
- from **June 1st 2015** all app updates will also have to support 64 bits.

Since AIR SDK version 16 we AIR developer have an easy way of ensuring that our apps and components support 64-bits on iOS. Below are the necessary steps for rebuilding your existing apps and ANEs for 64-bit support.

Step 1: Update your AIR SDK

1.1. Download the latest AIR SDK from Adobe:

- either get the final release;
- or the newest beta from Adobe Labs.

Note: if you use Flash Builder and/or the Flex SDK, which I'm guessing you might, if you picked up this book, you need the AIR SDK for Flex Developers, located at the bottom of the two pages linked above.

1.2. Make a backup copy of your AIR SDK folder.

1.3. Overlay the new SDK over the old one:

1.3.1. If you are a **Windows** user, unzip the download and copy the contents over your AIR SDK.

1.3.2. On Mac you can do the same and ask Finder to merge folders or, you can copy the archive in the root of your AIR SDK and run the following command in the Terminal:

```
sudo tar jxvf air16_sdk_sa_mac.tbz2
```

Step 2: Update your app descriptor

If you are a veteran AIR developer, you are used to this, but here it is just in case. Open your app descriptor file – usually named **your-app-name-app.xml** and found in your project's **src/** folder and make sure the namespace at the top of the file points to the SDK version you downloaded:

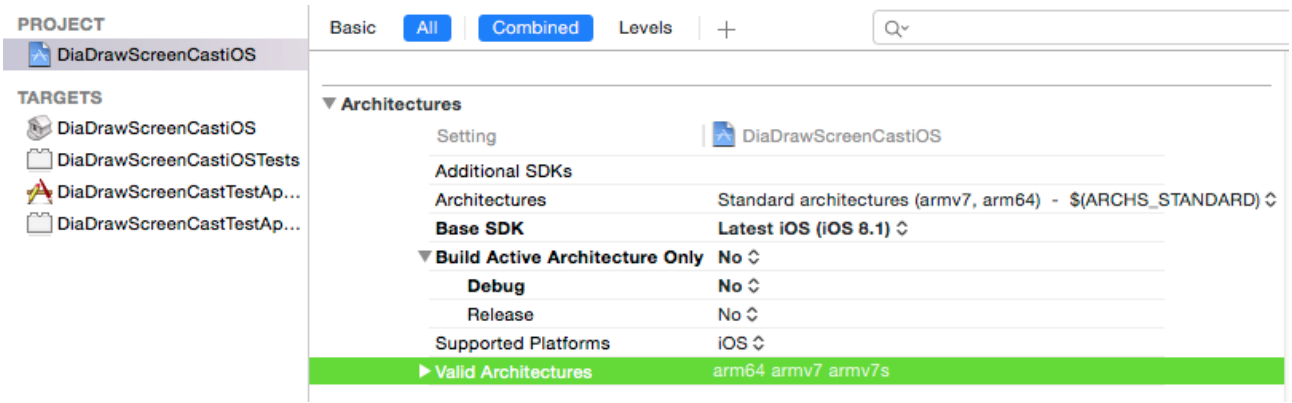
```
<application xmlns="http://ns.adobe.com/air/application/17.0">
```

Step 3: Rebuild your iOS ANEs

Any native extensions for iOS that your app uses will also need to be built with 64-bit support. If you have the source code for these ANEs, this is what you do:

3.1. Make sure you are using Xcode 6 and iOS SDK 8.

3.2. In your **Xcode** project, go to **Build Settings** > **Architectures** and first make sure that **Architectures** is set to **Standard architectures (armv7, arm64)**, then set **Build Active Architecture Only** to **No**. Without the last change the compiler will default to building a binary that supports only one architecture, which matches the device that you have connected at the moment or the simulator version you have made active – good idea for a debug build, but not for release.



Now make a clean build of your ANE. Here is a reminder for how to do that the hard way or the single-click way.

Step 4: How do you know if your app is now universal/64-bit?

… before submitting it to the Apple App Store, that is. There are a couple of things you can check.

First, if you build your app with AIR 16, you should NOT get an error message like this:

```
Error: Apple App Store allows only universal applications. "libMultiplatformANETemplateLib.a" is not a universal binary. Please change build settings in Xcode project to "Standard Architecture" to create universal library/framework.
```

Or like this:

```
[exec] Error: libCameraLibiOS.a are required to have universal iOS libraries. Please contact the ANE developer(s) to get the same.
[exec] Result: 12
```

Then, to double-check, do the following:

4.1. Rename your **.ipa** file to **.zip** and unzip it.

4.2. Use **lipo** in the Mac Terminal:

```
lipo path_to_your_unzipped_ipa/Payload/your_app_name.app/your_app_name -info
```

You want to see a message like this:

```
Architectures in the fat file: your_app_name are: armv7 arm64
```

4.3. You can also use the **file** command on Mac:

```
file path_to_your_unzipped_ipa/Payload/your_app_name.app/your_app_name
```

This should result in a message like this one:

```
Mach-O universal binary with 2 architectures
your_app_name (for architecture armv7): Mach-O executable arm
your_app_name (for architecture arm64): Mach-O 64-bit executable
```

Ta-da! Your app is now up to date with the 64-bit requirements.

Bonus chapter:

Using iOS frameworks in your ANE

When you develop your ANE or app, you may want to save time by using third party components. Or you may need to include parts of the iOS SDK to enable certain functionality that AIR doesn't link with by default.

A lot of the time these additional components are bundled as .framework files. A .framework bundle contains a linkable library, headers and any associated resources.

The ways you use iOS frameworks and third party frameworks in your ANEs are different and we will look at them in turn.

Using additional iOS frameworks in your ANE

Using third party frameworks in your ANE

Bonus chapter:

7 things you need to know about Ant scripts

1. What is Ant and where do I get it?
2. How do I write an Ant Script?
3. How do I run an Ant Script?
4. How do I define a default target?
5. How do run multiple targets in a specific order?
6. Got it. Now show me a complete Ant Script
7. What can I use to edit Ant scripts?

Where to go from here

First of all, thank you for sticking with this eBook!

We hope that you found it useful the way we meant it to be: giving you just what you need to know just when you need to know it.

With that said, this book is far from being a collection of everything there is to know about iOS native extensions under the sun, so here is a list of resources of additional information we found useful when we were learning how to make native extensions.

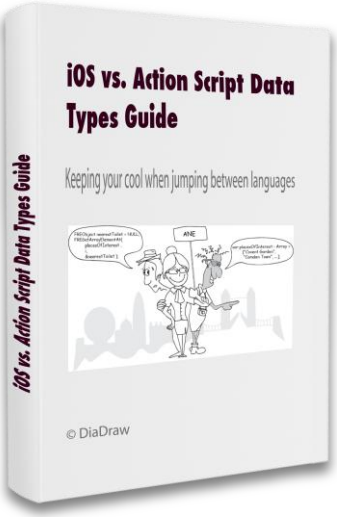
Useful online materials

- Check out the free online tutorials on EasyNativeExtensions.com and have a look at the free and commercial ANEs in our shop.

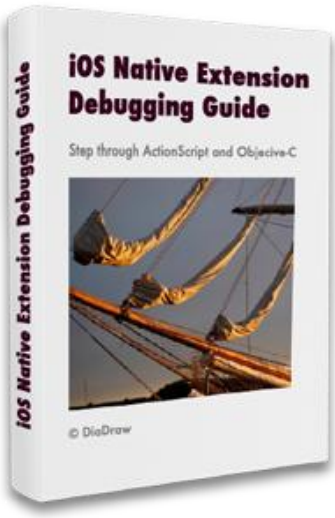
- The Adobe Flash Platform online manuals
    - How to Extend Your Mobile AIR Applications Using Native Extensions, an Adobe TV presentation by Olver Goldman, presented at MAX 2011
    - Beautiful Code: Notes from a developer's workbook, Rajorshi Ghosh's online blog on all things Adobe

- 20 tips for creating AIR Native Extensions for iOS, a blog post by Richard Lord

Want more?



- For a library of conversion functions that will save you time passing data between Objective-C and ActionScript download our iOS Data Types Guide: keeping your cool when jumping between languages.



- For easy recipes on how to debug the ActionScript and the native code of your iOS native extensions, get our iOS Native Extension Debugging Guide.

Thank you!

Thank you for picking this eBook and investing the time and money in it! We hope you found it as useful as you were hoping it would be. If you have a few minutes, **we would love to hear what you thoughts**.

Leave a comment on our blog or drop us an e-mail at office@diadraw.com. We read each and every message!

Other ways to get in touch and share

@DiaDrawCom on Twitter
https://www.facebook.com/DiaDraw on Facebook
DiaDraw on LinkedIn

Radoslava:

I loved writing this eBook. Every argument with the Adobe and Apple developer tools was worth getting into. I especially enjoyed having Hristo test each and every line of code or setup instruction, then doing a rewrite and asking him to start over… I mean, what sister worth her salt wouldn't relish that!

Hristo, thank you for every hour you put in diligently testing!

Hristo:

An eBook is much like a software project: it takes lot of effort to produce and there are people involved along the way who make sure that the final product looks and feels as promised. I am happy for the opportunity to work with Radoslava, testing, re-testing and striving to ensure a smooth experience for you. I hope that we've helped you build some great mobile apps.

P. S. Make sure to drop us a line and show off the next app you release!