WebKit Plug-In Programming Topics

Apple Applications > Safari

2008-10-15

Contents

Listings

Introduction to WebKit Plug-in Programming Topics

Web browser plug-ins are compiled bundles that help extend the content types supported by common web browsers. Installed locally on a computer, they can run code native to the user's operating system and provide a powerful way to expand on standard web content.

Who Should Read This Document?

This document is designed for a number of different audiences:

- If you are a Cocoa and WebKit developer, you should read about the WebKit plug-in architecture and learn how compiled plug-ins operate within WebKit-based applications, including Safari.
- If you are a developer who is concerned with cross-platform compatibility for your software, but who also wants to deploy special content via a web browser, you should read about the Netscape-based plug-in architecture and learn how it is supported in a variety of browsers.
- If you are a web content developer, you should read about both plug-in architectures and learn how to integrate their features into custom plug-ins to support your content.

Note: Safari supports the latest web standards, which may serve your needs more easily than a plug-in. If you are developing a plug-in to embed audio or video, you can instead take advantage of Safari's support for the `<audio>` and `<video>` tags in HTML 5. Client-side storage is available in Safari through the Storage and SQL APIs, also defined in HTML 5. If you are developing a rich user interface, Safari supports CSS transforms, transitions, and animations. More information on Safari and CSS can be found in *Safari CSS Reference*.

Organization of This Document

The topic contains the following articles:

- "About Web Browser Plug-ins" (page 9) describes the benefits of web browser plug-ins and how they are integrated into common browsers. It also discusses the advantages of disadvantages of both plug-in models, and how to deploy plug-ins on computers and web sites.
- "Creating Plug-ins with Cocoa and WebKit " (page 13) describes how to use the WebKit-based plug-in architecture to develop and deploy web browser plug-ins for Safari and WebKit-based applications.
- "Creating Plug-ins with the Netscape API" (page 19) describes how to use the Netscape plug-in architecture to develop and deploy web browser plug-ins across multiple browsers and platforms.

See Also

There are lots of helpful resources available to guide you through plug-in development.

- Read the *WebKit Objective-C Framework Reference* for the full WebKit plug-in reference and the reference detailing the WebKit-scripting environment bridge.
- Read the *WebKit Objective-C Programming Guide* for tips on good WebKit application design and how the web scripting environment can access WebKit methods and properties (and vice versa).
- On your hard drive, `/Developer/Examples/WebKit/` contains sample code for both the Netscape and the WebKit version of the movie player plug-in.
- Mozilla's Plug-ins Project discusses the cross-browser Netscape API and also includes lots of sample code.
- Plug-in Detections discusses how to tune your web content to detect plug-ins (if registration did not solve the problem).

About Web Browser Plug-ins

Web browser plug-ins are considered extensions to existing web browsers. By installing them locally on your machine, you can “teach” your web browsers to support alternative content types—perhaps even custom types you design yourself—or to perform additional tasks that a ready-made browser cannot do. For example, Macromedia Flash animations are not supported natively by any browsers. By installing their plug-in, though, you expand the capabilities of your browser to accept, interpret, and display the animation directly within the main content view.

The WebKit framework natively supports two different types of browser plug-ins. One is the Netscape-style plug-in, based off a common cross-platform API. The second is the WebKit–based plug-in, developed in Objective-C and supported by all WebKit-based applications.

Netscape-Style Plug-ins

Netscape-style plug-ins are written using a cross-platform C API and can be compiled in Mac OS X into either native Mach-O or classic PEF format. Though the API currently supports both formats, Mac OS X natively supports the Mach-O style, and you will find that your plug-in will run much faster if compiled as a Mach-O binary. In the future, Netscape plug-in API s will continue to support the Mach-O format; no such guarantee can be maintained for plug-ins compiled in the PEF format. You can also develop and debug Mach-O plug-ins in Xcode, but not PEF plug-ins. Once compiled, the plug-ins can be installed and used in most web browsers.

The original Netscape plug-in architecture was integrated into Netscape 2.0 in 1996. Since then, the API has since been adopted by most common web browsers, including Safari. It has built-in support for onscreen drawing, various types of event handling, and networking functions.

In addition to the core plug-in functionality, Apple has extended the capabilities of the Netscape-style plug-ins. Starting with the WebKit framework bundled with Safari 1.3 (on Mac OS X version 10.3) or Safari 2.0 (on Mac OS X version 10.4), the Netscape-style plug-ins can also perform scripting functions.

The new Netscape-style plug-in scripting environment allows plug-ins to access scripting languages such as JavaScript (including accessing script elements such as a web page’s Document Object Model). It also allows scripting languages to access and control elements of the plug-in.

WebKit–Based Plug-ins

WebKit-based plug-ins are written using Objective-C API that is unique to WebKit. Although the API is not cross-platform, any plug-in created in this fashion can be used in Safari and all other WebKit-based applications. This style of plug-in is easy to develop and significantly cuts down on the amount of code you have to write, because it harnesses all the premade API and interface technology available to Cocoa applications.

Because WebKit-based plug-ins are based on subclasses of Cocoa’s NSView, this style also offers support for onscreen drawing and event handling.

Apple has also provided these types of plug-ins with access to the enclosing scripting environment; a script can call methods and read properties from the plug-in, and vice versa.

When you create a WebKit-based plug-in you need to compile it as a universal binary. For more information, see *Universal Binary Programming Guidelines, Second Edition*.

What Kind Of Plug-in Should I Develop?

Because the WebKit framework provides for two different and distinct plug-in technologies, you may be asking yourself, what technology should I use?

You should use the Netscape-style plug-in architecture if:

- Your plug-in needs to be supported on multiple Mac browsers, not all of which use the WebKit framework.
- Your plug-in will be deployed on multiple platforms. Because the API is cross-platform, you would just have to compile the same code for Windows or Linux.

Otherwise, using the WebKit–based plug-in architecture is probably the right plan for you. You will save lots of development and debugging time, and your total lines of code will be much lower than for a Netscape plug-in. However, you will be able to deploy these plug-ins only on Mac OS X, for applications that use the WebKit framework (including Safari).

Registering Your Plug-in

Your plug-in will need to register itself with the application that uses it, so that the application knows what content types you support. Your plug-in bundle needs to contain this important registration information. In addition to setting the correct registration information for your plug-in, you must set the correct `CFBundlePackageType`. The default `CFBundlePackageType` when you create a new WebKit plug-in in Xcode is `BNDL`, this must be changed to `WBPL` to be recognized by WebKit.

There are two different ways of storing this registration information. One is with an `Info.plist` file stored within the plug-in bundle. This is an easy-to-edit and easy-to-maintain way to register your content types, but is supported only by applications based on the WebKit (no matter in what style you wrote the plug-in). The other way to register the information—and this method is required if you want other browsers on the Mac platform to support your plug-in—you also have to include a Carbon resources file.

The `Info.plist` file contains important information. Let's use an example from the WebKit movie plug-in, which you will create in "Creating Plug-ins with Cocoa and WebKit " (page 13). First, you need to register a description of the plug-in (see Listing 1).

Listing 1 Plug-in description in the Info.plist file

```
<key>WebPluginDescription</key>
    <string>Simple WebKit plug-in that displays movies</string>
```

Next, you'll need to register the MIME types that your plug-in supports (see Listing 2).

Listing 2 Registering MIME types in the Info.plist file

```
<key>WebPluginMIMETypes</key>
    <dict>
        <key>video/x-webkitmovie</key>
        <dict>
            <key>WebPluginExtensions</key>
            <array>
                <string>mov</string>
            </array>
            <key>WebPluginTypeDescription</key>
            <string>QuickTime Movie</string>
        </dict>
    </dict>
```

Finally, your plug-in needs a useful name for the application to call it by (see Listing 3 (page 11)).

Listing 3 Name That Plug-in

```
<key>WebPluginName</key>
    <string>WebKit Movie Plug-in</string>
```

Installing Your Plug-in

Plug-ins can be stored in one of two places on a Mac OS X system:

- Plug-ins stored in `/Library/Internet Plug-ins` can be shared by all users on the computer.
- Plug-ins stored in `~/Library/Internet Plug-ins` will be available only to the user whose home directory contains them.

In addition, the WebKit-based plug-ins can be stored inside the bundle of *any* application that uses the WebKit, by storing it in:

```
    AppName/Contents/Plug-ins
```

where `AppName` is the actual application's executable bundle.

Plug-ins are generally reloaded on each application start.

Deploying Your Plug-in

Content that your plug-ins will view needs to be embedded within HTML. Most browsers do this with an `EMBED` tag, but others require the `OBJECT` tag. For maximum compatibility, you can tune your page to support both. The example in Listing 4 is specific to the QuickTime plug-in provided by Apple, and you can tune these parameters to your own mode of business.

Listing 4 Embedding a movie into an HTML page

```
<OBJECT CLASSID="clsid:02BF25D5-8C17-4B23-BC80-D3488ABDDC6B"
 WIDTH="160"
 HEIGHT="144"
 CODEBASE="http://www.apple.com/qtactivex/qtplugin.cab">
 <PARAM name="SRC" VALUE="sample.mov">
 <PARAM name="AUTOPLAY" VALUE="true">
 <PARAM name="CONTROLLER" VALUE="false">
<EMBED SRC="sample.mov"
WIDTH="160"
HEIGHT="144"
AUTOPLAY="true"
CONTROLLER="false"
PLUGINSPAGE="http://www.apple.com/quicktime/download/">
 </EMBED>
</OBJECT>
```

The variables for the Active X controls will change based on your plug-in's behavior. Read Apple's *HTML Scripting Guide for QuickTime* tutorial for more information.

Creating Plug-ins with Cocoa and WebKit

You can write browser plug-ins with the native WebKit plug-in API. Written in Objective-C, WebKit-based plug-ins are supported only by WebKit-based applications and cannot be ported to other platforms. The API is extremely simple, so many fewer lines of code are required to deploy a WebKit plug-in versus a Netscape one and you can use Xcode and Interface Builder to design and implement a plug-in's functionality.

Introduction To WebKit Plug-ins

WebKit plug-ins are based on core Cocoa API. The plug-in itself is simply an instance of an NSView, a common class in many other Objective-C applications. It provides a cornucopia of features, including the management of events such as mouse and keyboard inputs. Your plug-in inherits these "for free." URL loading is also inherited, via NSURLConnection. You can access WebKit classes through the plug-in's WebFrame and the browser scripting environment through the WebKit WebScriptMethods protocol.

Becoming A Plug-in

For the plug-in to act like a standard web browser plug-in, it needs to conform to the `WebPlugIn` informal protocol. This protocol has just one required constructor method, `plugInViewWithArguments:`, which your NSView subclass should implement.

Optional methods you can implement include:

- `webPlugInInitialize`, which is called just after the plug-in is created and allows you to perform any prestartup actions in the plug-in.
- `webPlugInStart`, which is called when the plug-in should begin doing whatever it has been designed to do.
- `webPlugInStop`, which is called to tell the plug-in to cease its usual actions.
- `webPlugInDestroy`, which is called to give the plug-in a chance to deallocate any objects or resources it may have created or retained.
- `webPlugInSetIsSelected:`, which is called when the selection state of the plug-in has changed, allowing you to do any custom drawing or actions based off that event.

These methods are implemented by the **container** of the plug-in; that is, they affect the web view that surrounds the plug-in:

- `webPlugInContainerLoadRequest:inFrame:` allows you to tell the browser to load a URL request into a given frame (or the container's frame itself).
- `webPlugInContainerShowStatus:` allows you to tell the container to print a status message to the browser's status bar.

- `webPlugInContainerSelectionColor` returns the color that the container should use to draw plug-in's selection state when it is selected.
- `webFrame` allows you to access the other WebKit elements of the container, such as its `WebView`.

Using Plug-in Scripting

The WebKit API allows your plug-ins to easily access a scripting environment (such as JavaScript) from the plug-in, and vice versa. Your plug-in can call JavaScript methods and read JavaScript properties, while your containing page can call methods from your plug-in from its JavaScript environment.

When the browser encounters your plug-in, it will use JavaScript to request the object representing your plug-in using `objectForWebScript`. The object that you return from that method represents the interface to your plug-in. This can be, but is not required to be, the same object as your plug-in. In that case, your implementation of `objectForWebScript` would simply look like:

```
- (id)objectForWebScript
{
    return self;
}
```

The object you return needs to have control over which of its methods should be visible to the scripting environment. In all likelihood, you don't want all of your methods exposed to the environment, which they will be, by default. To counteract this, implement these methods:

- `webScriptNameForSelector:` returns the name that a given selector should inherit so that it can be called from the JavaScript environment. The default renaming scheme (to prevent against namespace conflicts) can lead to confusing method names in the scripting environment, so you should make a habit of rewriting the names of all your exposed methods. For example, if you had an Objective-C method called `startMovieAtBeginning`, you might want it to reflect its own name in the scripting environment instead of going through a rewrite. An implementation example would look like:

  ```
  (NSString *)webScriptNameForSelector:(SEL)selector {
      if(selector == @selector(startMovieAtBeginning)) {
          return @"startMovieAtBeginning";
      }
      return nil;
  }
  ```

- `isSelectorExcludedFromWebScript:` lets the scripting environment know whether or not a given Objective-C method in your plug-in can be called from the scripting environment. A common mistake first-time plug-in developers make is forgetting to implement this method, causing the plug-in to expose no methods and making the plug-in unscriptable. As a security precation this method returns `YES` by default exposing no methods. You want to expose only methods that you know are secure, to do this the function should return `NO`. You may only want to export a couple of your Objective-C methods to JavaScript. In this example, the plug-in's `play` method can be called from JavaScript, but any other method cannot::

  ```
  + (BOOL)isSelectorExcludedFromWebScript:(SEL)selector {
      if(selector == @selector(play)) {
          return NO;
      }
      return YES;
  ```

```
    }
```

Similarly, you want to give the scripting environment access to all of your properties. The syntax is very similar for restricting those:

- `webScriptNameForKey:` should be implemented to return a more human-readable name for a method to the scripting environment.
- `isKeyExcludedFromWebScript:` allows you to selectively expose properties to the scripting environment.

Implementing a Plug-in

In this example, you create a QuickTime movie plug-in. This is a powerful example, because it requires very few lines of code and yet provides a useful extension to a web browser or WebKit application.

First, you need to create the view class. In this case, you use Cocoa's built-in NSMovieView and subclass it to create your `PlugInMovieView` (see Listing 1).

Listing 1 PlugInMovieView header (PlugInMovieView.h)

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

@interface PlugInMovieView : NSMovieView
{
    NSDictionary *_arguments;
    BOOL _loadedMovie;
    BOOL muted;
}

- (void)setArguments:(NSDictionary *)arguments;

@end
```

Now you can write the implementation. You first need to conform to the `WebPlugIn` protocol, by implementing `plugInViewWithArguments:` (see Listing 2). Create an instance of your movie view, assign it the arguments passed into your method, and return it. Notice that an accessor method is being used to set the arguments—this is good Cocoa coding style.

Listing 2 Returning your plug-in's view

```
+ (NSView *)plugInViewWithArguments:(NSDictionary *)arguments
{
    PlugInMovieView *movieView = [[[self alloc] initWithFrame:NSZeroRect] 
autorelease];
    [movieView setArguments:arguments];
    return movieView;
}
```

Now that you've returned the view, you need to make a decision. Do you have any operations to perform on initialization? In the case of NSMovieView, you can set a movie's controller to be visible (or not) and also specify whether or not you'd like the user to be able to adjust its size. In this case, you should show the controller but prevent the user from resizing the movie in the frame—the most common layout for embedded movies (see Listing 3).

Listing 3 Initializing the movie plug-in

```
- (void)webPlugInInitialize
{
    [self showController:YES adjustingSize:NO];
}
```

From the enclosing container, nestled in an embed tag, you'll receive a URL pointing to a movie. This will arrive in one of the keys specified by the *arguments* dictionary that you set in Listing 2. Use that URL to load and play the movie (see Listing 4).

Listing 4 Loading and playing a movie from a URL

```
- (void)webPlugInStart
{
    if (!_loadedMovie) {
        _loadedMovie = YES;
        NSString *URLString = [[_arguments objectForKey:WebPlugInAttributesKey]
 objectForKey:@"src"];
        if ([URLString length] != 0) {
            NSURL *baseURL = [_arguments objectForKey:WebPlugInBaseURLKey];
            NSURL *URL = [NSURL URLWithString:URLString relativeToURL:baseURL];
            NSMovie *movie = [[NSMovie alloc] initWithURL:URL byReference:NO];
            [self setMovie:movie];
            [movie release];
        }
    }

    [self start:self];
}
```

Eventually, all good things must come to an end, and so shall your plug-in. This will be announced by a call to `webPlugInStop`. You should take the opportunity to stop the movie from playing (see Listing 5).

Listing 5 Stopping the movie

```
- (void)webPlugInStop
{
    [self stop:self];
}
```

You've just implemented a fully functional WebKit movie-playing plug-in. You could build this code, install the plug-in, and have your own working QuickTime player embedded in Safari or a WebKit-based application. However, you might want to add a little more flair and use a form—with HTML buttons—to play and pause the movie. It just takes a few more lines of code (see Listing 6).

Listing 6 Opening the plug-in to JavaScript

```
+ (BOOL)isSelectorExcludedFromWebScript:(SEL)selector
{
    if (selector == @selector(play) || selector == @selector(pause)) {
        return NO;
    }
    return YES;
}

+ (BOOL)isKeyExcludedFromWebScript:(const char *)property
{
    if (strcmp(property,"muted") == 0) {
        return NO;
    }
    return YES;
}

- (id)objectForWebScript
{
    return self;
}

- (void)play
{
    [self start:self];
}

- (void)pause
{
    [self stop:self];
}
```

You only had to add two extra methods, `play` and `pause`, so that the buttons in the interface could be tied to public methods. Then you exposed those methods to the JavaScript scripting environment.

If you want to explore further, this example is available at:

```
    /Developer/Examples/WebKit/WebKitMoviePlugIn
```

Creating Plug-ins with the Netscape API

Netscape-style plug-ins are programmed in C, and, provided that you are building in Mach-O form, can be developed and debugged with Xcode.

Using Plug-in Scripting

The scripting capabilities of Netscape-style plug-ins are provided by extensions onto the original plug-in specification. They allow a browser (through JavaScript) to access and control elements of the plug-in and its content, and allow the plug-in to access the enclosing web page and its content through the plug-in script interface.

When a plug-in is loaded, the browser calls the JavaScript method `NPP_GetValue`, which should return NPClass and NPObject instances that represent your plug-in. The NPClass defines the interface between the plug-in and the scripting environment, while NPObject represents your custom instance of that class which can then be used by the scripting environment. Thus, if you want your plug-in to be scriptable, you need to return the appropriate NPObject by reference in your own implementation of `NPP_GetValue`.

A good demonstration of accessing plug-ins from JavaScript, as well as all the other concepts in creating a Netscape plug-in, can be found at::

```
/Developer/Examples/WebKit/NetscapeMoviePlugIn
```

Document Revision History

This table describes the changes to *WebKit Plug-In Programming Topics*.

Date	Notes
2008-10-15	Added information regarding alternatives to plug-in development.
2006-12-05	Updated code sample for webScriptNameForSelector.
2006-04-04	Made minor editorial corrections throughout.
2006-02-07	Added information to the isSelectorExcludedFromWebScript function.
2006-01-10	Added information about universal binaries.
2005-08-11	Corrected typos in sample code and updated link to the QuickTime Active X Plugin tutorial.
2005-04-29	New document that explains how to develop and deploy browser plug-ins based on the Web Kit architecture.