# WebKit Plug-In Programming Topics

(Retired Document)

# Contents

2

# Listings

SwiftObjective-C

# Introduction to WebKit Plug-in Programming Topics

**Important:** This document may not represent best practices for current development. Links to downloads and other resources may no longer be valid.

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 plug-in architecture used by WebKit 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:** As of Safari 5.1, the legacy WebKit plug-in architecture is no longer supported. Going forward, you must convert WebKit plug-ins to Netscape-style plug-ins or Safari Extensions.

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 HTML5. For more information, see the *Safari HTML5 Audio and Video Guide* in the Safari Reference Library.

Client-side storage is available in Safari through the Storage and SQL APIs, also defined in HTML5. 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* in the Safari Reference Library.

## Organization of This Document

The topic contains the following articles:

- About Web Browser Plug-ins (page 7) describes the benefits of web browser plug-ins and how they are integrated into common browsers. It also discusses the advantages and disadvantages of both plug-in models, and how to deploy plug-ins on computers and web sites.

- Creating Plug-ins with the Netscape API (page 9) describes how to use the Netscape plug-in architecture to develop and deploy web browser plug-ins across multiple browsers and platforms.

- Creating Legacy Plug-ins with Cocoa and WebKit (page 16) describes the legacy WebKit-based plug-in architecture.

## See Also

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

- 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).

- 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).

# 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.

The WebKit framework natively supports the Netscape-style plug-in API, which is based off a common cross-platform API. This API is described futher in Creating Plug-ins with the Netscape API (page 9).

---

**Historical Note:**  Previous versions of Safari supported a WebKit–specific plug-in model based on Objective-C. This plug-in model is no longer supported in Safari for two main reasons:

- WebKit plug-ins are closely tied to WebKit in such a way that the WebKit plug-in model could not be practically modified to support out-of-process execution (which is desirable for both security and stability reasons).

- Most plug-ins in WebKit are based on the Netscape-style plug-in model because it supports non-WebKit-based browsers, and it allows you to share most of your code with the Linux and Windows versions of your plug-in.

Thus, you should exclusively use the Netscape-style plug-in architecture going forward.

---

## Installing Your Plug-in

Plug-ins can be stored in one of two places on an 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 1 is specific to the QuickTime plug-in provided by Apple, and you can tune these parameters to your own mode of business.

**Listing 1**        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 the Netscape API

Netscape-style plug-ins are written using a cross-platform C API, and, provided that you are building in Mach-O form, can be developed and debugged with Xcode. These plug-ins are compatible with a wide range of web browsers.

The original Netscape plug-in architecture was integrated into Netscape 2.0 in 1996. Since then, the API has 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 OS X version 10.3) or Safari 2.0 (on OS X version 10.4), the Netscape-style plug-ins can also perform scripting functions.

The 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.

In OS X v10.5 and earlier, Netscape-style plug-ins can be compiled in OS X into either the Mach-O or PEF (CFM) binary format (although CFM is PowerPC-only). Beginning in OS X v10.6, Netscape-style plug-ins should be updated to use a 64-bit binary to work with 64-bit instances of Safari or other WebKit clients. To do this, they must be built as 32/64-bit multi-architecture binaries using the Mach-O file format.

Though the API supports both formats, 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, WebKit will continue to support Netscape plug-ins built with the Mach-O format; no such assurance can be given for plug-ins compiled in the PEF (CFM) 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.

## 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 `NPP_GetValue` callback in your plug-in, which returns a retained `NPObject` structure that represent your plug-in. This `NPObject` structure contains an pointer to an `NPClass` structure. The `NPClass` structure contains a series of callbacks that define the interface between the plug-in and the scripting environment. The `NPObject` instance represents an instance of that plug-in that can then be used by the scripting environment.

If you want your plug-in to be scriptable, you need to return the appropriate retained `NPObject` by reference in your `NPP_GetValue` callback function.

> **Note:** In current versions of WebKit and all non-WebKit-based browsers, you must retain the `NPObject` instance upon return by doing the following:
>
> ```
> browser->retainobject((NPObject*)obj);
> ```
>
> In versions of WebKit prior to version 420 (Safari 3 and earlier), the objects returned are retained by the browser. To avoid memory leaks in older browsers, you should check the WebKit version and avoid retaining returned objects when your plug-in is loaded by prior versions of WebKit.

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 in the Examples folder of the WebKit source code. See http://www.webkit.org/ to learn how to download the WebKit source code.

## Core Graphics and Core Animation Drawing Models

Safari 4.0 provides two new drawing models: Core Graphics (Quartz 2D) and Core Animation (only in OS X v10.5 and later). These drawing modes are strongly recommended going forwards, and if you move your plug-in to contain a 64-bit slice, that slice must use these drawing models. (See Transitioning a Netscape-Style Plug-in to 64-bit (page 14) for more information.)

Most of the effort in using these drawing models comes from learning Core Graphics and Core Animation themselves. To learn about Core Graphics, read *Quartz 2D Programming Guide*. To learn about Core Animation, read *Core Animation Programming Guide*.

Once you understand how to draw things using Core Graphics or Core Animation, you can enable these drawing models in your `NPP_New` function as follows:

- Add support in your code for the Cocoa event model. Core Graphics and Core Animation are not supported when using the Carbon event model.

- Check to see if the host browser supports the drawing model with the following code:

```
NPBool supportsCG = false;
NPError error = browser–>getvalue(instance,
    NPNVsupportsCoreGraphicsBool,
    &supportsCG);
```

> **Note:** If you cannot find a drawing model that both the application and your plug-in supports, your plug-in should return NPERR_INCOMPATIBLE_VERSION_ERROR.

- Set the browser drawing model with the following code:

```
if (err == NPERR_NO_ERROR && supportsCG) {
    error = browser–>setvalue(instance,
        NPNVpluginDrawingModel,
        (void *)NPDrawingModelCoreGraphics);
    if (err == NPERR_NO_ERROR && supportsCG) {
        /* Set state flags as needed to
            tell your own code to use Core Graphics
            drawing in the future. */
    }
}
```

After you have done these two things, the browser fills the `window` field of the `NPWindow` structure with an `NP_CGContext` structure. This structure contains two fields, `context` and `window`, which are defined as follows:

```
typedef struct NP_CGContext
{
    CGContextRef context;
    WindowRef window;
} NP_CGContext;
```

The `context` value is a Core Graphics drawing context suitable for Quartz 2D drawing. For more information on how to use this context, read *Quartz 2D Programming Guide*. The window is a reference to an `NSWindow` object.

To obtain the bounds for your plug-in's drawing region, do the following in your `NPP_SetWindow` callback:

```
NPError setwindow_cb(NPP instance, NPWindow* npw) {

    ...


    NP_CGContext *npcontext = npw.window;
    CGContextRef context = npcontext.context;
    CGRect boundingBox = CGContextGetClipBoundingBox(context);


    ...

```

The Core Animation model is similar, but reversed. If you set you set `NPNVpluginDrawingModel` to `NPDrawingModelCoreAnimation`, your `NPN_GetValue` callback must provide a retained Core Animation layer to the browser when it queries the `NPPVpluginCoreAnimationLayer` variable.

As with the Core graphics model, you can find out if the browser supports the Core Animation drawing model by checking the value of the `NPNVsupportsCoreAnimationBool` variable. For example:

```
NPBool supportsCA = false;
NPError error = browser->getvalue(instance,
    NPNVsupportsCoreAnimationBool,
    &supportsCA);
```

## Out-of-Process Plug-Ins

Beginning in OS X v10.6, on 64-bit-capable computers, Netscape-style plug-ins execute in an out-of-process fashion. This means that each Netscape-style plug-in gets its own process separate from the application process. This design applies to all Netscape-style plug-ins. It does not apply to WebKit plug-ins, nor at present to WebKit-based applications when running in 32-bit mode.

Out-of-process execution is used for several reasons:

- Stability—crashes in plug-ins no longer crash the entire browser.

- Security—vulnerabilities in a plug-in can't read or alter data or code in other plug-ins or in the application itself.

For your plug-in to work correctly in this new environment, you may need to make some changes to your code, depending on how it interacts with the rest of the system beyond the browser.

- **Avoid direct data structure access.** If you directly access application data structures (menus, other windows, etc.), your plug-in will fail. All access to browser data structures should be performed through approved APIs. If no API exists, file a bug and request that one be added.

- **Draw when asked.** Your plug-in *must* respond to `drawRect` event.

- **Draw only when asked.** Any drawing your plug-in performs within the graphics context that Safari provides should take place in response to an `update` event (Carbon event model) or a `drawRect` event (Cocoa event model). The behavior of drawing performed at other times is undefined.

  To provide support for animation, video, and other drawing outside the context of `update` or `drawRect` events, you should move your code to use the Core Animation drawing model, described in Core Graphics and Core Animation Drawing Models (page 10).

- **Get all events through the plug-in API.** Because keyboard and mouse events are happening in a different process, if you use Carbon or Cocoa calls to obtain data from the event stream, you will not receive any events.

  It is safe to use Carbon or Cocoa calls to obtain global system state information that is outside the scope of the event stream, such as mouse position, modifier keys, and other similar information.

- **Use platform APIs sparingly.** Wherever possible, you should use new plug-in APIs to do what you need. If no such APIs exist, file bugs requesting them.

  The plug-in API provides additional APIs for scheduling timers, opening contextual pop-up menus, and so on.

  > **Note:** Calls to browser scripting functions work normally. WebKit transparently handles the interprocess communication for you.

- **Avoid creating windows.** The intent is for plug-ins to operate within the browser window. Although some plug-ins have historically done so, creating windows in your plug-in is not recommended. If you need to maintain separate windows, you should consider starting a separate application.

# Transitioning a Netscape-Style Plug-in to 64-bit

Beginning in Snow Leopard, Safari is moving to a 64-bit process. This section describes how to adapt a Netscape-style plug-in so that it works when compiled as a 64-bit plug-in. It describes only the changes specific to Netscape-style plug-ins, providing links to other documents for general 64-bit porting information.

Before you begin, do the following:

- If your plug-in is built using PEF (CFM) compilers (CodeWarrior, for example), you must move to a modern toolchain that generates Mach-O binaries. There are no 64-bit compilers available that support PEF. The easiest way to do this is to transition your project to Xcode. For more information, read *Porting CodeWarrior Projects to Xcode* .

- You must convert your GUI code from Carbon to Cocoa (or other graphical APIs that are supported in a 64-bit environment). For more information on supported and unsupported Carbon APIs, read *64-Bit Guide for Carbon Developers* . Then, read *Cocoa Fundamentals Guide* to get started learning Cocoa. The preferred APIs for drawing are described further in Core Graphics and Core Animation Drawing Models (page 10).

- If your plug-in uses any Cocoa APIs, read *64-Bit Transition Guide for Cocoa* to learn what other Cocoa-specific changes you need to make to this code.

- Read *64-Bit Transition Guide* to learn about 64-bit changes that apply to all software written in C-based languages.

Once you have read these documents, you need to make a few additional changes specific to Netscape-style plug-ins (at least when compiling 64-bit versions of your plug-in). These changes are as follows:

- **Use Cocoa Events.** In a 64-bit environment, you must convert your code to use the Cocoa event model because the Carbon event model is not supported. This requires several changes:

  - Use the `NPCocoaEvent` data type with the Cocoa event model instead of the `NPEvent` data type. The `NPCocoaEvent` data type provides neatly broken out fields containing various pieces of information about the event, and is fairly self-explanatory.

  - Disable the Carbon model test code (including the test to see if the Carbon model is supported) when compiling for 64-bit by wrapping the relevant code with the `#ifndef __LP64__` and `#endif` C preprocessor directives.

    The `NPEventModelCarbon` constant (for use with the `NPNVpluginEventModel` variable) and the `NPNVsupportsCarbonBool` constant (for use with `NPN_GetValue`) are not defined at all in a 64-bit environment. By using the C preprocessor directives above, you prevent the compiler itself from ever seeing the intervening lines of code, thus avoiding a compile failure.

  Bring up your Cocoa event model code in 32-bit first. This makes it possible to test the Cocoa event code in isolation without having to worry about bugs specific to the 64-bit environment.

Do not remove your Carbon event model code, however. Although the Cocoa event model is supported in 32-bit WebKit beginning in OS X v10.6, it may not be supported in other browsers that use Netscape-style plug-ins, nor in earlier versions of WebKit.

For maximum backwards compatibility and cross-browser compatibility, you should check the `NPNVsupportsCarbonBool` and `NPNVsupportsCocoaBool` variable using `NPN_GetValue` to determine which event models the browser supports. If the browser supports both event models, you should use the Cocoa event model. Otherwise, use the Carbon event model.

For testing purposes, simply hard-wire your test to always use the Cocoa event model.

- **Replace QuickDraw.** The QuickDraw drawing model is not supported. You must rewrite any QuickDraw code using another drawing model such as the Core Graphics or Core Animation drawing modes.

  As a result, the `NPDrawingModelQuickDraw` constant (for use with the `NPNVpluginDrawingModel` variable) and the `NPNVsupportsQuickDrawBool` constant (for use with the `NPN_GetValue` function) are not defined. Thus, any code that tests for the availability of the QuickDraw drawing model must be wrapped with `#ifdef __LP64__` and `#endif` C preprocessor directives.

  > **Note:** You can take the same approach to replacing QuickDraw code that you too with the Carbon event model—deploy first on 32-bit using a test to see which models are supported, temporarily tweak that test to report that QuickDraw isn't available (for testing purposes), then test and deploy your new drawing code on 32-bit. Use C preprocessor directives to skip the test entirely when compiling 64-bit so that only the Core Graphics or Core Animation drawing model code is seen by the compiler.

  Similarly, the `NPQDRegion` and `NP_Port` data types are not defined.

  In the `NP_CGContext` data type, the `window` field may be either a `WindowRef` instance or an `NSWindow` instance in 32-bit WebKit depending on the drawing model chosen. Because Carbon drawing APIs are not supported, these fields always contain an `NSWindow` instance in 64-bit WebKit.

  For more information about the Core Graphics and Core Animation drawing models, see Core Graphics and Core Animation Drawing Models (page 10).

Once you have accounted for these differences, you should be ready to compile your plug-in with a 64-bit slice. For more information on compiling code for 64-bit, fixing truncation bugs, and various other relevant topics, read *64-Bit Transition Guide for Cocoa* and *64-Bit Transition Guide*.

# Creating Legacy Plug-ins with Cocoa and WebKit

SwiftObjective-C

In versions of OS X prior to version 10.7, WebKit supported plug-ins written to the native WebKit plug-in API. Because WebKit plug-ins are so closely tied to browser data structures, these plug-ins cannot run in a separate process. Thus, to improve security and stability, Safari in OS X v10.7 no longer supports this model.

Newer versions of the Netscape plug-in model and the Extensions model supersede the functionality of WebKit plug-ins. For more information on the Netscape plug-in model, read Creating Plug-ins with the Netscape API (page 9).

## Introduction To Legacy 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 prior to OS X v10.7.

WebKit plug-ins are based on core Cocoa API. Written in Objective-C, WebKit-based plug-ins are supported only by WebKit-based applications and cannot be ported to other platforms.

A WebKit plug-in itself is simply an instance of the `NSView` class, which is a common class in many other Objective-C applications. It provides a wide range of features, including management of events such as mouse and keyboard input. Your plug-in inherits these "for free." For example, you can easily load URLs using an `NSURLConnection` object.

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. You can also access WebKit classes through the plug-in's `WebFrame` object and the browser scripting environment through the WebKit `WebScriptMethods` protocol.

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* .

# Becoming A Plug-in

For a 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.

> ⚠️ **Warning:** WebKit-style plug-ins are unsupported in OS X v10.7 and later. You should port your plug-in to the Netscape plug-in model instead. See Creating Plug-ins with the Netscape API (page 9) for more information.

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.

> ⚠️ **Warning:** WebKit-style plug-ins are unsupported in OS X v10.7 and later. You should port your plug-in to the Netscape plug-in model instead. See Creating Plug-ins with the Netscape API (page 9) for more information.

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. By default, Objective-C methods are not exposed as JavaScript methods. To expose some of your instance methods in JavaScript, override the following 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 precaution this method returns `YES` by default exposing no methods. You should expose only methods that you know are secure; to export a method, this function should return `NO` for that method's selector. You may only want to export one or two Objective-C methods to JavaScript. In the following example, the plug-in's `play` method can be called from JavaScript, but other methods cannot:

```
+ (BOOL)isSelectorExcludedFromWebScript:(SEL)selector {

    if(selector == @selector(play)) {

        return NO;

    }

    return YES;

}
```

Similarly, you may want to give the scripting environment access to certain properties in your plug-in object. 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.

> ⚠️ **Warning:**  WebKit-style plug-ins are unsupported in OS X v10.7 and later. You should port your plug-in to the Netscape plug-in model instead. See Creating Plug-ins with the Netscape API (page 9) for more information.

First, you need to create the view class. In this case, you use Cocoa's built-in `NSMovieView` class 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 the `NSMovieView` class , 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
```

on a computer running OS X v10.5 or earlier with Xcode Tools 3.1.3 or earlier installed.

## Transitioning a WebKit Plug-In to 64-bit

⚠️ **Warning:** WebKit-style plug-ins are unsupported in OS X v10.7 and later. You should port your plug-in to the Netscape plug-in model instead. See Creating Plug-ins with the Netscape API (page 9) for more information.

Beginning in OS X v10.6, Safari is a 64-bit WebKit-based application. Thus, all WebKit plug-ins must be updated to include 64-bit slices in their universal binaries in order to work with Safari.

Because WebKit plug-ins are inherently Cocoa-based code, relatively few changes should be needed to make your code work in a 64-bit version of Safari (beyond recompiling it with different architecture settings). You do not need to make any changes specific to WebKit plug-ins.

Before you begin, read the documents *64-Bit Transition Guide* and *64-Bit Transition Guide for Cocoa*. The main transition guide describes compiler settings and changes you must make to port any code to a 64-bit environment. The Cocoa transition guide describes additional changes you must make that are specific to a Cocoa environment.

Once you have read these documents, the porting a WebKit plug-in is essentially the same as porting a Cocoa application. This is a four step process:

- Change your compile settings to build a 3-way or 4-way universal binary (with 32-bit PowerPC, 32-bit Intel, 64-bit Intel, and optionally 64-bit PowerPC slices) and recompile. Fix any problems that prevent compilation.

> **Note:** Safari does not use the 64-bit PowerPC slice of plug-in binaries regardless of what
> operating system or Safari version you use. However, some third-party WebKit-based applications
> may use it in 64-bit mode in OS X v10.5.

- Make the data type changes described in *64-Bit Transition Guide* and *64-Bit Transition Guide for Cocoa*.

- Enable additional compiler warnings to help you find and fix truncation bugs, then fix as many of the warnings as possible.

- Track down and fix any remaining bugs that prevent correct operation.

## 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 Legacy Plug-ins with Cocoa and WebKit (page 16). First, you need to register a description of the plug-in (see Listing 7).

**Listing 7**      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 8).

**Listing 8**      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 9 (page 25)).

**Listing 9**      Name That Plug-in

```
<key>WebPluginName</key>

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

# Document Revision History

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

| Date | Notes |
| --- | --- |
| 2011-05-17 | Updated for OS X v10.7. |
| 2009-03-13 | Added information about 64-bit WebKit plug-ins in OS X v10.6. |
| 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. |