Safari Extensions Development Guide

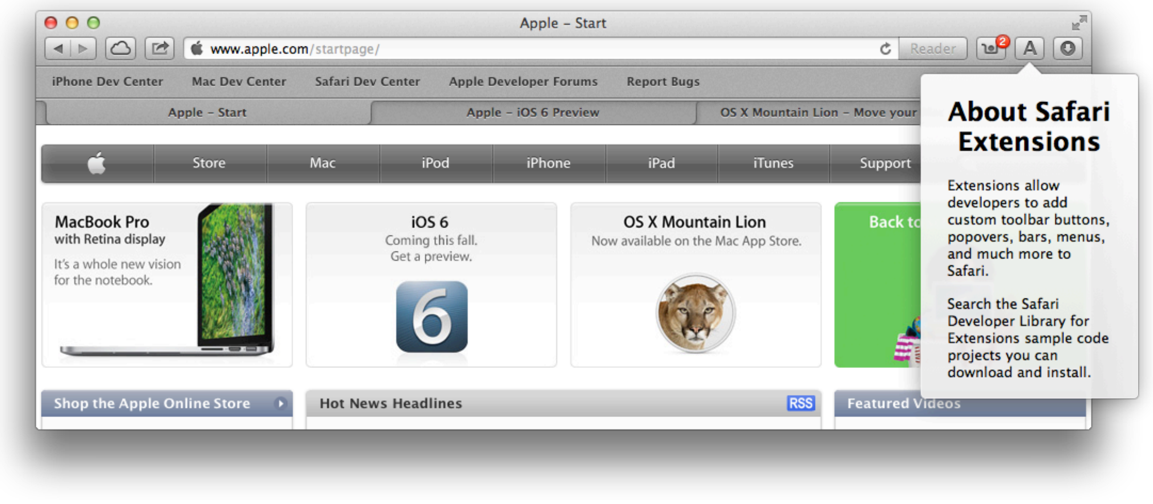
Developer

Contents

Figures and Listings

About Safari Extensions

Safari extensions provide a way for you to add features to the Safari browser. You can add custom buttons to the Safari toolbar, create bars of your own, add contextual menu items, display content in bars or tabs, and inject scripts and style sheets into webpages. In Safari 5.1 and later, you can add menus and popovers to toolbar items.



About Safari Extensions

Extensions allow developers to add custom toolbar buttons, popovers, bars, menus, and much more to Safari.

Search the Safari Developer Library for Extensions sample code projects you can download and install.

You write Safari extensions using HTML, CSS, and JavaScript, with support for HTML5 and CSS3. A JavaScript API for extensions allows you to interact with the browser and web content in ways that scripts normally can't.

Important: To develop extensions for Safari, you need to sign up for the Safari developer program online, at http://developer.apple.com. You need a signed certificate before your extension can be installed.

Safari extensions are supported in Safari 5.0 and later on the desktop (Mac and Windows). Safari extensions are not currently supported on iOS.

At a Glance

Safari extensions let you add persistent items to Safari—controls, menus and menu items, local or web-based content, and scripts that modify the content Safari presents.

What’s the Difference Between an Extension and a Plug-in?

A plug-in can add support for media types to a browser. An extension can add many different features.

Extensions and plug-ins both expand a browser’s capabilities. Plug-ins let browsers display media that the browser can’t display natively or provide a particular media player experience. Extensions personalize and enhance the browser itself and can interact with HTML web content.

A plug-in can’t interact with webpages except to display media of specific MIME types. A plug-in cannot add features to Safari, such as toolbar buttons or contextual menu items.

A plug-in is a binary file that interfaces with the browser but is essentially an app in itself—the browser hands off specific media types to the plug-in to handle.

An extension is a collection of HTML, JavaScript, and CSS files that the browser uses to expand its feature set. Extensions allow you to reformat webpages, block unwanted sites or unwanted material, display RSS feeds and other data in a bar or window, and do thousands of other things that plug-ins can’t do.

Extensions Have Their Own JavaScript API

Extensions have access to a special JavaScript API that lets them access the Safari app and web content. This API is documented in *Safari Extensions Reference* .

Relevant Chapter: “Extensions Overview” (page 13)

Extensions Run in a Sandbox

Safari Extensions run in their own “sandbox” or container area. The designated container area for Safari Extensions is the execution of normal HTML, CSS, and JavaScript within the Safari app, and the specific JavaScript APIs documented here and *Safari Extensions Reference* .

Important: Launching an installed Safari plug-in from an extension by using the HTML `<object>` or `<embed>` tags is strongly discouraged, but not prohibited. Any other use of an extension as a method to launch code that executes outside of the Safari app is prohibited.

You Create Extensions Right in Safari

You create extensions using Extension Builder, which is built into Safari 5.0 and later. Open Extension Builder, tell it to create an extension folder, drag your HTML, JavaScript, and CSS style sheets into the folder, fill out the fields in Extension Builder, and you’re ready to go.

The main ingredients of an extension are:

- **Global HTML page**—code that's loaded once, when Safari launches or when your extension is enabled. This is the ideal place to put the code for buttons in the Safari toolbar, extension menus, or contextual menus. This page is never displayed; it's just for logic.
- **Content** (HTML, CSS, JavaScript, media)—HTML content and interactive controls that you display in popovers, extension bars, or in tabs as full-page content. Extension bar and popover files have access to the Safari app and can include logic for menus or toolbar items.
- **Menu Items** (labels, images)—items that appear in extension menus that you define, or are added to Safari's existing contextual menus.
- **Injected scripts**—scripts to be injected into browser content. These scripts can read, modify, add to, or delete content.
- **Injected style sheets**—user style sheets that can modify the display of web content by overriding or adding to the styles normally applied.
- **Icon image**—the icon for your extension.

Relevant Chapter: "Using Extension Builder" (page 25)

You Can Define User Settings in Extension Builder

Your extension can have its own user settings, accessible to the user in the Extensions pane of Safari preferences. You define the settings, user interface items, and default values using Extension Builder.

There is also a settings API similar to HTML5 local storage for accessing and modifying settings programmatically. You can use encrypted settings for security.

Relevant Chapter: "Settings and Local Storage" (page 99)

Debug Your Extension with Safari's Built-in Tools

You can use the Safari developer tools to help debug your extension. The developer tools report HTML errors and profile JavaScript, log messages to the console, and let you interactively set breakpoints, get variable values, and call functions. The debugging tools are supported for extension bars, global HTML pages, and injected scripts. Each extension bar and global page has its own console.

Relevant Chapter: "Debugging Extensions" (page 106)

Update Your Extension Automatically from the Web

Safari provides a method to support checking for updates to an extension automatically: the Update Manifest. You specify a web address, and Safari periodically compares the installed version of your extension with the latest version on your website. If your website has a newer version, Safari offers the user an update.

Relevant Chapter: "Updating Extensions" (page 110)

Prerequisites

You need to be familiar with HTML, JavaScript, and the basics of CSS. Familiarity with HTML5 and CSS3 is helpful. To add a button to the toolbar, you need to be able to create an image with an alpha channel (transparency).

See Also

- *Safari Extensions Reference* —the JavaScript classes, methods, and properties that only Safari extensions can use.
- *Safari DOM Additions Reference* —the classes, methods, and properties that Safari exposes to JavaScript that are not available in all browsers.
- *Safari Web Inspector Guide* —the built-in web development tools that come with Safari.
- *Safari Extensions Conversion Guide* —a guide to converting extensions written for other browsers.

Extensions Overview

Extensions are a way for you, as a developer, to add features to Safari.

You write Safari extensions using HTML, CSS, and JavaScript, with support for HTML5 and CSS3. Safari exposes a set of methods and properties to JavaScript for extensions to use, letting your extension do things that scripts normally can't.

Safari Extension Certificates

To develop extensions for Safari, you first need to sign up for the Safari Developer Program online at http://developer.apple.com. You need to join the program and obtain a certificate before your extension can be installed. There is a link to the Developer Certificate Utility on the main Safari developer webpage. Your extension does not load unless you obtain and install a certificate from Apple.

One certificate supports all your extensions. The certificate functions on multiple computers if both the *certificate* and *private key* are imported from the originating computer to each new computer.

> **Important:** If you move a certificate to another computer you must export the private key from the original computer. Import the private key into the new KeyChain. The private key and the certificate must match to be valid.

When the certificate nears expiration, create and download a new certificate from the developer site. Because the two certificates carry the same developer ID, your existing extensions recognize the new certificate. After implementing the new certificate, revoke old certificates.

What Your Extension Can Do

Safari extensions let you add persistent items to Safari, such as controls, local or web-based content, and scripts that modify web-based content.

- You can add buttons to the Safari toolbar.
- You can add menus and submenus to extension-defined toolbar buttons.
- You can add custom toolbars (extension bars).

- You can add items to Safari's contextual menus.
- You can display HTML content in an extension bar, in a popover, in its own tab, or inject the content into webpages.
- You can detect the availability of the Safari article reader, enter it programmatically, and inject scripts and style sheets into it.
- Your extension can run invisibly in the background.
- You can modify and reformat web content by applying scripts and style sheets.

  Your scripts and style sheets can be applied either universally or selectively, using whitelists and blacklists of URL patterns to determine which webpages they should be applied to.
- You can post notifications to Notification Center, OS X's systemwide notification system, as you would from a webpage using HTML5 notifications. Your extension does not need explicit permission from the user. For more information, see *Notification Programming Guide for Websites* .

To see examples of Safari Extensions, visit the Safari Extensions Gallery (https://extensions.apple.com/).

The Extension Parts List

An extension starts as a folder. Depending on what you want your extension to do, you put some or all of the following items into the folder:

- **Global HTML page**—An HTML page containing support code for your extension, typically JavaScript. This page is never displayed, but it can contain HTML elements that are used by other parts of your extension. The global HTML page is loaded once—when the app launches or your extension is installed or enabled—and has access to the Safari app API. This is the right place to code buttons for the Safari toolbar, extension menus, or contextual menu items.
- **Content files**—HTML, CSS, and JavaScript content to display in extension bars, popovers, full-page tabs, or to inject into webpages by creating an `iframe`.

  Popovers and extension bars have access to the Safari app-level API and can also contain code for toolbar items or menu items.

  Content files for tabs can be hosted on remote web servers but it is recommended that they reside in your extension package. Content files for popovers or extension bars *must* reside in your extension package.
- **Injected scripts**—JavaScript files to be injected into browser content. These scripts can read, modify, add to, or delete content, and can be applied selectively to webpages using URL patterns.
- **Injected style sheets**—User style sheets that can modify the display of web content by overriding or adding to the styles normally applied. These style sheets can also be applied selectively using URL patterns.

- **Icon**—The icon for your extension. It should be at least 64x64 pixels. Name the image file `Icon.png`. The icon image is scaled up or down as needed. You can also supply images that are optimized for display at particular sizes.

  If you supply multiple icons for display at different sizes, name the 64x64 icon `Icon-64.png`. For best results, include an `Icon-48.png` and `Icon-32.png` for optimized display at smaller sizes. Optimize the appearance of your icons on high-resolution displays by including an `Icon-96.png` and an `Icon-128.png` as well.
- **Other images and media**—Any other images or media your extension needs, such as images that appear in menus.

Extension Architecture

You can think of extensions as being divided into two parts: a part that interacts with the Safari app, and a part that interacts with web content.

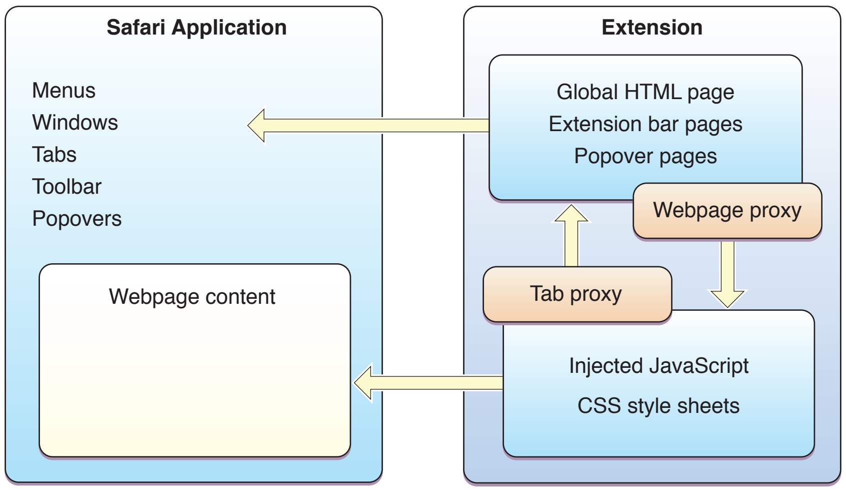
The part of an extension that interacts with the Safari app resides in any of your extension's global HTML page, extension bar pages, or popover pages. The part that interacts with web content resides in JavaScript files or CSS style sheets that are injected into content pages.

The division between these parts is strict, but you can send messages between them using proxies. If the global HTML page or an extension bar page needs to act on web content, it sends a message via the webpage proxy, where an injected script can act on it.

Similarly, if an injected script needs to make use of code in the global HTML page or an extension bar, it can send a message via the tab proxy.

The extension architecture is illustrated in Figure 1-1.

Figure 1-1 The extension architecture



An extension does not necessarily need to have both of these parts—an extension can operate only on the Safari app or only on web content. For example, a toolbar button to close a window or insert a tab would interact only with the app, while a style sheet that reformats websites into black text on a white background would operate only on web content.

The Safari Extensions JavaScript API

In addition to the usual JavaScript methods, extensions have access to a special JavaScript API that lets them access the Safari app and web content. The full API is documented in *Safari Extensions Reference*, but this section covers the main things you need to know.

Classes and Properties

The Safari extensions API includes several classes such as `SafariBrowserWindow`, `SafariBrowserTab`, and `SafariWebPageProxy`, representing, respectively, a window, a tab, and the webpage loaded in a tab. You rarely, if ever, use the actual class names in your code, however. Instead, your extension JavaScript uses the `SafariNamespace` object, `safari`, followed by a chain of properties. For example:

- `safari.application.activeBrowserWindow` returns the active instance of `SafariBrowserWindow`.
- `safari.application.activeBrowserWindow.activeTab` returns an instance of `SafariBrowserTab`.
- `safari.application.activeBrowserWindow.activeTab.page` returns an instance of `SafariWebPageProxy`.

As usual in JavaScript, there is more than one way to address a particular object and the chain of properties goes both ways—a browser window has a `tabs` property representing its tabs, for example, and each tab has a `browserWindow` property representing its parent window.

The Application and Extension Objects

The `SafariApplication` object allows you to work with windows and tabs, and to respond to commands from toolbar items and contextual menu items (also known as shortcut menu items). For example, you open a new browser window like this:

```
safari.application.openBrowserWindow();
```

The `SafariExtension` object allows you to add and delete buttons, menu items, scripts, and style sheets from your extension. For example, the following code snippet adds a simple black-and-white style sheet to the injected contents of your extension:

```
var bw = "body { color:black !important; background:white !important }" ;
safari.extension.addContentStyleSheet(bw);
```

You can access the `SafariApplication` and `SafariExtension` classes from your extension's global HTML page or from an extension bar or popover. The classes are accessed as `safari.application` and `safari.extension`.

Web Content Interaction

Scripts that are injected into web content can access the DOM of webpages they are injected into, allowing them to read and modify the content. Injected scripts use the normal JavaScript API—`getElementsByTagName()`, `innerHTML`, and so on—but because they are injected into a webpage, they have the privileges of a script loaded from the same domain the content comes from. In other words, a script injected by an extension can do anything the website author's own scripts can do.

You can also designate style sheets as injected content. Injected style sheets are treated as user style sheets, as defined by the W3C. This means that they can override styles applied by the webpage's author if they are declared important. For example, to override the `body` element's background color, you could declare:

```
body { background: #ffffff !important }
```

The style cascades in the following order:

1. Your injected style sheet's normal declarations are applied.
2. The website author's style sheet's normal declarations are applied.

3. Styles declared as important in the website author's style sheets are applied.
4. Styles declared as important in your injected style sheets are applied, overriding any previous definitions (you have the last say).

Events—Commands, Messages, and Proxies

You respond to events by installing an event listener—a function that handles a specified type of event. If you've written JavaScript event handlers before, you know that you can install an event listener on the event's target element or any of its parent elements with the window at the top of the tree. The window receives events for all the elements on the webpage. In Safari extensions, there is a higher level still: the app, which receives events for all the open windows.

JavaScript programmers normally have only one place to put an event handler: the script for the webpage. In a Safari extension, there are several places where you can put an event handler, such as your global HTML page, an extension bar or popover page, or an injected script. Different events can be handled in different places.

There are several types of events in the Safari extensions API, but there are three fundamental event types you should be familiar with right away: `"command"`, `"validate"`, and `"message"` events.

Command events are generated when the user clicks an extension's toolbar item or chooses an extension's menu item (including contextual menu items). To handle commands, install a listener function for `"command"` events. Inside your listener function, test the command name for commands you are responsible for. Add a listener function by calling `addEventListener("command", function, capture)`.

Here's an example of adding a `"command"` event handler to the app:

```
safari.application.addEventListener("command", myCmdHandler, false);
```

Validate events are sent for important browser events—such as opening a window or loading a page. Prior to any command events or menu displays, these events ensure the menu items and commands are valid. You can respond to a `"validate"` event by disabling your toolbar item or menu item, modifying what it does, or by doing nothing if the command should be executed normally.

You can respond to `"command"` and `"validate"` events in either your global HTML page (recommended) or in an extension bar or popover.

Message events are your way to pass information between parts of the extension. Messages are sent using `dispatchMessage(messageName, data)`. You listen for messages by installing a listener function for `"message"` events: `addEventListener("message", functionName, false)`.

The message API is accessible from all parts of an extension—the global HTML page, popovers, extension bars, and injected scripts.

Proxies are used to support message passing across the app/content boundary. A proxy object represents a pair of objects on different sides of the boundary. There is a `page` proxy object (class `SafariContentWebPage` / `SafariWebPageProxy`) for sending messages to injected scripts and a `tab` proxy object (class `SafariBrowserTab` / `SafariContentBrowserTabProxy`) for sending messages to an extension bar or to the global page.

How to Create Extensions

Extensions are created using Extension Builder, which is built into Safari 5.0 and later. Enable the Develop menu in the Advanced pane of Safari preferences. Then choose Show Extension Builder in the Develop menu.

Note: Extensions were introduced in Safari 5.0 and were disabled by default, so in Safari 5.0 you must enable extensions in the Develop menu before you can show Extension Builder.

Extensions are enabled by default in Safari 5.0.1 and later.

An extension consists of an extension package—a signed, compressed folder with the `.safariextz` extension, containing all your extension's files and a generated `plist` file that tells Safari how your extension is organized and what it does.

Note: The Safari extension package is an OS X bundle. You don't need to understand bundles to create extensions, but you may find it helpful. To learn more about bundles and the `plist`, see *Bundle Programming Guide*.

To create an extension, first make an extension folder by clicking the + button in Extension Builder and choosing New Extension. Then create the HTML, CSS, JavaScript, and media files you need and put them in the folder. The folder has the `.safariextension` extension when it is created.

Use Extension Builder to specify details about the structure and behavior of your extension and to build an extension package. Clicking Build creates a compressed, installable version of your extension with the `.safariextz` extension. For details, see "Using Extension Builder" (page 25).

Here's a more detailed description of the things you put into the extension folder:

Global HTML Page

Your extension can have a global HTML page, but it is not mandatory. This page is loaded only when Safari first loads your extension, but it is never displayed. It exists as a container for JavaScript. You can add JavaScript to your global page inline, or include it in a separate file or files in your extension.

If you are adding items to the main Safari toolbar, it's generally best to write a global HTML page to specify what the toolbar items do. But you can also specify what the items do in a extension bar file. For details, see "Adding Buttons to the Main Safari Toolbar" (page 52). Toolbar buttons can also invoke extension menus or popovers. The logic for extension menus generally belongs in the global HTML file as well. The logic for popovers can reside in the global HTML file or the popover file itself. For details, see "Adding Extension Menus" (page 59) and "Adding Popovers" (page 64).

If you are adding items to Safari contextual menus, it's generally best to write a global HTML page and specify what the menu items are and what they do, but again, you can also specify contextual menu items and actions in an extension bar or popover file. For details, see "Adding Contextual Menu Items" (page 69).

Putting the code for toolbar items, pop-up menus, and contextual menu items in your global page is more efficient than putting it in an extension bar file. This is because extension bar files are reloaded every time a window is opened, whereas the global file is loaded only once during the app's lifetime.

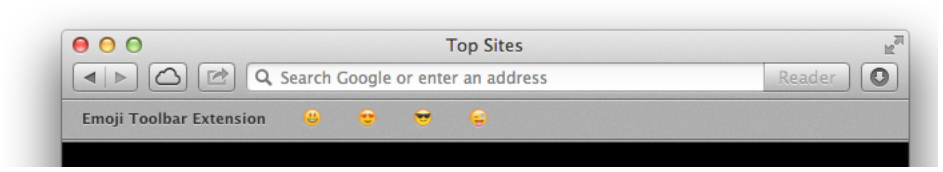
If your injected scripts use a large amount of code or data, it should be moved to the global HTML page, so time isn't spent reloading large blocks of code or data each time the user opens a webpage. Injected scripts can't call functions defined in your global page directly, but injected scripts can pass messages to the global page, and the message handler in the global page can call other functions. For details, see "Messages and Proxies" (page 87).

Extension Bar Files

Extension bars are toolbar-sized strips added to the Safari frame—below the bookmarks bar and above the tab bar—and dedicated to a particular extension. There can be multiple extensions with bars installed, and multiple bars per extension. If more than one extension bar exists, they are stacked. An example of an extension bar is shown in Figure 1-2 (page 21).

Each extension bar has a label that is listed in the View menu (the View menu is hidden by default in Windows, but can be accessed through the gear button) and the menu item can be toggled to show or conceal each bar in the stack.

Figure 1-2 Extension bar example



You can use extension bars to add controls to Safari or to display other content, such as a stock ticker, weather forecast, flight information, or headlines. Extension bars are only 30 pixels tall, so content that needs a taller display space should be shown in a popover, in its own tab, or injected into the browser content instead.

Extension bar files can access the Safari app to do things like opening and closing windows and tabs, loading URLs, responding to Safari toolbar items, and responding to menu choices in extension menus or contextual menus. Extension bar files cannot access the content layer to manipulate content loaded in a browser tab directly, however; for that you need to use injected scripts or styles (see "About Safari Extensions" (page 9)). Your extension bars can send messages to and receive messages from your injected scripts.

You create extension bars using HTML (also CSS, JavaScript, and any media files). You don't need to do anything special in the HTML to have your content displayed in an extension bar—just tell Extension Builder which HTML files are sources for extension bars.

If your extension bar uses images or other media, they can be included in the extension package or loaded from the web at runtime. It is strongly recommended that you use local media whenever possible.

Extension bar files are loaded each time Safari opens a browser window, creating an instance of the bar in every window, so if your extension bar has code or data that needs to load only once, you should put that material in a global HTML page instead.

If you want to create an extension bar, see "Adding Extension Bars" (page 35).

Popover Files

If your extension needs more space to display content than fits comfortably in an extension bar, or if you want the content to appear only when the user summons it, you can create a popover—an HTML file that displays in a pop-up window, then disappears when the user changes focus (by clicking in another window, for example).

Popover files have the same access and permissions as extension bar files, but display as pop-up windows instead of persistent bars. A popover is displayed in response to the user clicking, or pressing and holding, a toolbar item defined by the extension.

An extension can have multiple popovers, but only one displays at a time. Each popover is an element in the `safari.extension.popovers` array. If popovers are specified in Extension Builder, each popover file loads once, when the extension launches. If a popover is created at runtime, the specified popover file is loaded then. There is only one instance of a popover, no matter how many windows are open.

In Safari 5.1 and later, you can add popovers and associate them with Safari toolbar items. For details, see "Adding Popovers" (page 64).

Injected Scripts and Style Sheets

You can have Safari inject scripts or style sheets that you provide into the webpages Safari loads. These injected scripts and styles can read and modify browser content.

Scripts can be specified as End Scripts (interpreted when the page's `onload` event occurs), or Start Scripts (interpreted before the page is parsed). Most scripts are End Scripts. Scripts that block unwanted content before it displays are the most common use for Start Scripts. You can have both Start Scripts and End Scripts.

Style sheets are applied as user style sheets, so normal declarations in them precede the webpage author's declarations in the cascade, but `!important` declarations are applied after the author's declarations, allowing user style sheets to override the webpage author's styles.

You can use URL patterns to decide which webpages your scripts and style sheets are applied to. Use URL patterns when creating a blacklist or a whitelist for your extension. The blacklist contains URL patterns for webpages you don't want to inject scripts or styles into. The whitelist contains URL patterns for webpages you do want your scripts and styles injected into. For details, see "The Extension Builder Interface" (page 26).

If you want to inject scripts into webpages, see "Injecting Scripts" (page 76).

If you want to apply user style sheets to webpages, see "Injecting Styles" (page 79).

The plist Files

The `Info.plist` file contains your extension's metadata. This includes the extension name, author, and version, as well as information about how your extension is organized—whether it has a global HTML page, extension bars, or injected scripts, and which files are used for what. If your extension has settings, they are also defined in a `plist` file—`Settings.plist`. `Settings.plist` is optional, but `Info.plist` is required. When someone talks about your extension's `plist` file, they generally mean `Info.plist`.

The `plist` files are created for you using Extension Builder, so you shouldn't need to do anything with them yourself. But to really understand how extensions work, you need to know the `plist` files exist. All the fields you fill out in the Extension Builder interface are stored in a `plist` file.

Security

Because extensions have privileges that allow them to go beyond what ordinary scripts can do, you need to be security conscious when writing extensions. The first level of security is provided by the certificate. The certificate ensures that an extension comes from a known source (you) and prevents malicious extensions from masquerading as your extension. It is your responsibility to prevent your extension from being taken over by a malicious script, however. Follow the guidelines in the following sections to prevent security breaches.

Don't Insert Imported Text Using innerHTML or document.write

It can be tempting to display HTML imported from the web using the `innerHTML` property, but it is dangerous to do so. It is equally dangerous to insert imported text into your extension using the `document.write` method. A malicious website can include a script that, when loaded into the HTML of your extension, gains the privileges of your extension. Furthermore, if your extension obtains the imported HTML using HTTP (as opposed to HTTPS), a forged DNS server, such as a Wi-Fi hacker, can substitute a malicious script for the requested item.

If you are displaying imported text, insert it into one or more paragraph elements using the `innerText` property instead of using the `innerHTML` property or the `document.write` method.

If you are displaying imported HTML, sanitize it by removing any material enclosed by dangerous tags, such as scripts or HTTP requests, before inserting the HTML into your extension. Use a whitelist of allowed tags and attributes, and remove any HTML that does not match the whitelist. Insert the HTML into your extension using DOM methods such as `appendChild`, which inserts only safe elements.

Don't Use eval with Imported Text

Using the `eval` method to parse imported data allows the data to be executed as code, enabling malicious scripts to execute from within your extension. Use the `JSON.parse` method instead.

Don't Use HTTP to Add HTML, CSS, or Scripts

Include any scripts, HTML, and CSS that your extension uses directly within your extension, or obtain them from a trusted source using HTTPS. If you import items using HTTP, a person with network access, such as a Wi-Fi hacker, can insert malicious scripts in place of the requested items.

Private Browsing

Private Browsing mode prevents Safari from storing cookies, browsing history, search history, caches, and AutoFill information. Your extension should not store any information about the user's actions when in Private Browsing mode.

In Safari 6.0 and later, you can check if Safari is in Private Browsing mode by querying `safari.privateBrowsing.enabled`, or by listening for activate and deactivate events that have the `SafariPrivateBrowsing` object as their targets. Private Browsing will not be enabled until after the activate event is completed, allowing extensions to do any necessary cleanup. See *SafariApplication Class Reference* for more information.

Using Extension Builder

You can use Extension builder to build, install, reload, and uninstall extensions. Extension Builder is built into Safari 5.0 and later.

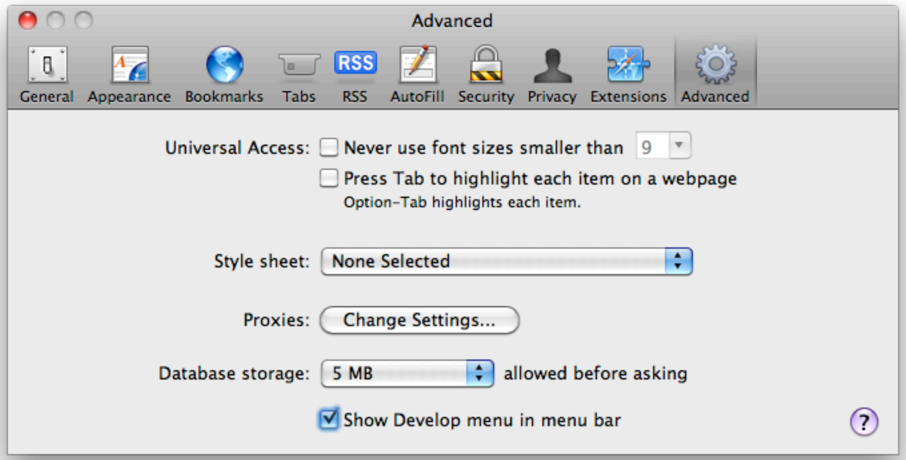
Before You Begin

Before you can build and install an extension, you need to install a developer certificate. You obtain a certificate by signing up for the Safari Developer Program at http://developer.apple.com. Install your certificate by double-clicking the certificate file. This launches Keychain Access on Mac, or the Certificate Import Wizard on Windows.

Opening Extension Builder

To access Extension Builder, first enable the Safari developer tools by clicking "Show Develop menu in menu bar" in the Advanced pane of Safari preferences, as shown in Figure 2-1. (To learn more about the Safari developer tools, see *Safari Web Inspector Guide*).

Figure 2-1 Enabling the developer tools



Universal Access: [ ] Never use font sizes smaller than 9
[ ] Press Tab to highlight each item on a webpage
Option-Tab highlights each item.

Style sheet: None Selected

Proxies: Change Settings...

Database storage: 5 MB allowed before asking

[✓] Show Develop menu in menu bar

Note: In Safari 5.0, extensions are turned off by default and must be enabled by choosing Enable Extensions in the Develop menu, so the Extensions icon is not initially shown in Safari preferences; otherwise in Safari 5.0.1 and later, extensions are enabled by default.

Next, choose Show Extension Builder from the Develop menu. If no previous work has been done in Extension Builder, the Extension Builder window is largely empty except for the + button. Don't worry; that's normal.

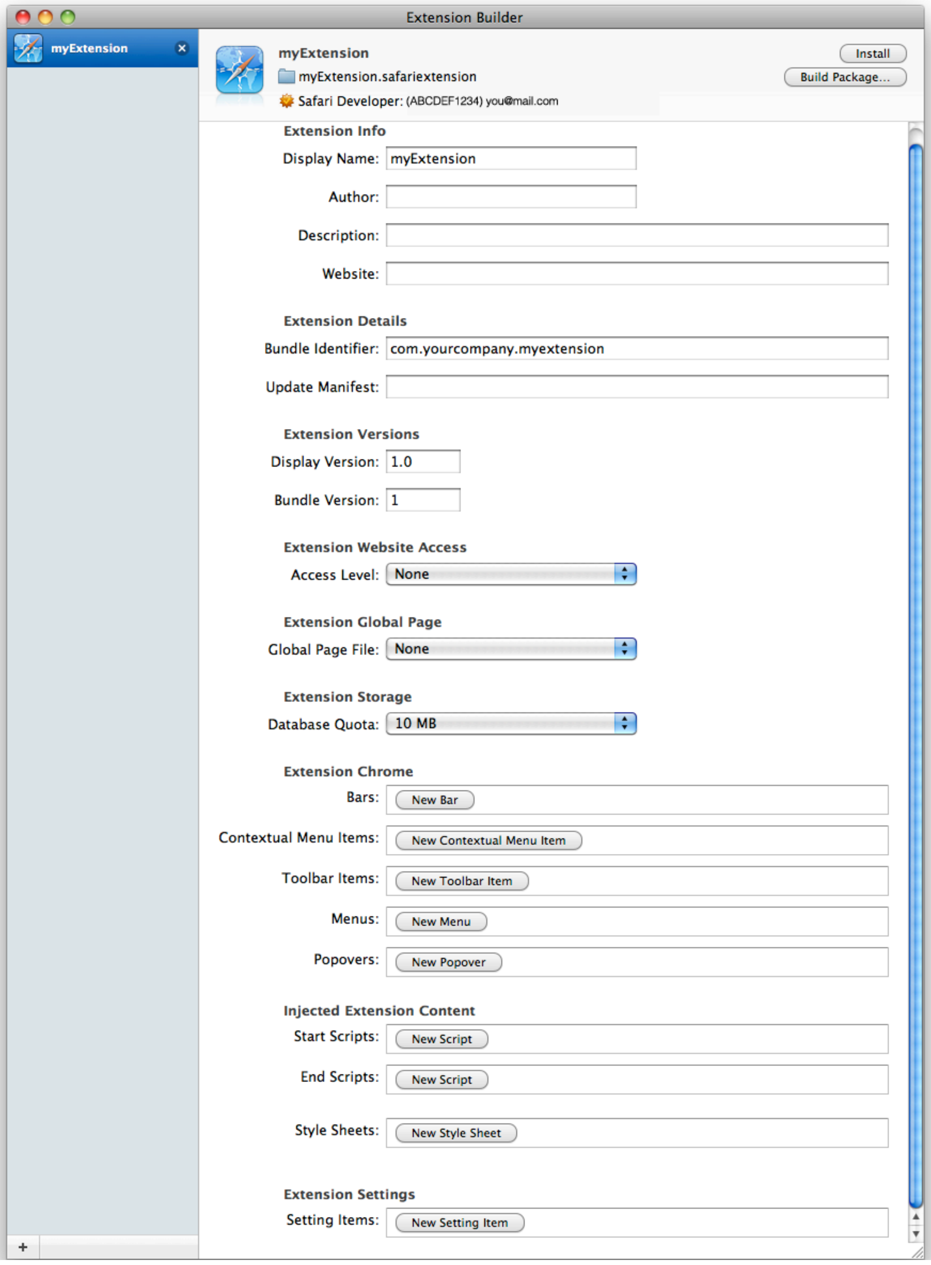
The Extension Builder Interface

Click the + button to create a new folder to hold your extension. You are prompted to either Add Extension (choose an existing extension folder) or create a New Extension. Choose New Extension. You're prompted to give the folder a name and location. The folder is created with the `.safariextension` extension.

A generic extension icon is displayed with a label derived from your folder name. Add your extension's HTML, CSS, JavaScript, and media files to this folder. If you include an `Icon.png` or `Icon-64.png` file, the icon changes to show it, otherwise the icon remains generic. Your certificate information is also displayed. The information is shown in the format: `Safari Developer: (Developer ID) email@address`.

Click your extension's icon to bring up the main Extension Builder interface, as shown in Figure 2-2.

Figure 2-2 Extension Builder interface



myExtension
myExtension.safariextension
Safari Developer: (ABCDEF1234) you@mail.com

Extension Info
Display Name: myExtension
Author:
Description:
Website:

Extension Details
Bundle Identifier: com.yourcompany.myextension
Update Manifest:

Extension Versions
Display Version: 1.0
Bundle Version: 1

Extension Website Access
Access Level: None

Extension Global Page
Global Page File: None

Extension Storage
Database Quota: 10 MB

Extension Chrome
Bars: New Bar
Contextual Menu Items: New Contextual Menu Item
Toolbar Items: New Toolbar Item
Menus: New Menu
Popovers: New Popover

Injected Extension Content
Start Scripts: New Script
End Scripts: New Script
Style Sheets: New Style Sheet

Extension Settings
Setting Items: New Setting Item

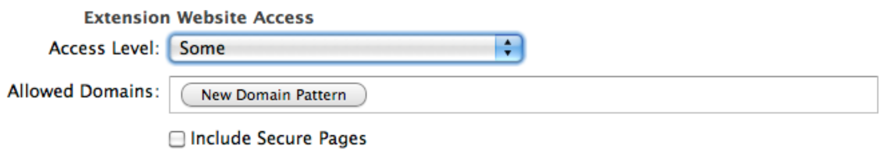
At the top of the window is your extension's icon, title, the name of the extension folder, and your developer ID. The following interactive fields are displayed below:

- Display Name—The visible name of your extension. *Required* .
- Author—Your name or your company name.

- Description—A brief description of what your extension does.
- Website—The website that users should visit for information and support.
- Bundle Identifier—OS X bundle identifiers are alphanumeric strings in reverse DNS format. Use your type of organization (com, gov, edu, org, and so on), your company name, and the extension name, separated by dots. For example, `com.MyCompany.myExtension`. *Required*.
- Update Manifest—The URL to use when checking for available updates. For more information, see "Updating Extensions" (page 110).
- Display Version—The displayed version number for your extension. *Required*.
- Bundle Version—The internal version number used by the operating system. One or more digits with period separators, such as 1 or 4.1.1. This is the version number Safari uses when checking for updates. *Required*.
- Extension Website Access—Use this field to restrict your extension's access to external websites. Your choices are as follows:
  - None—Your extension cannot access webpages by injecting scripts or style sheets.
  - Some—Your extension has access to webpages specified in the Allowed Domains field. If this field is left blank, your extension has no website access.
  - All—Your extension can access webpages from any domain.

  If you choose Some or All, you can further choose to allow your extension access to secure sites (HTTPS URLs) or not, as shown in Figure 2-3.

  Figure 2-3 Access restrictions

  
  Extension Website Access
  Access Level: Some
  Allowed Domains:

  ☐ Include Secure Pages

  When listing domains and pages your extension is allowed to access, you can use the asterisk as a wildcard character. For more information on access and permissions, see "Access and Permissions" (page 95).
- Global Page File—A page loaded once, when Safari loads your extension. This page is not displayed. It is intended for JavaScript functions that handle response to commands, messages, and other events.
- Database Quota—The space you want to allocate for HTML5 client-side database storage for your extension. For more information, see "Settings and Local Storage" (page 99).

- Bars—An extension bar is space below the bookmarks bar and above the tab bar reserved for your extension. If you want to display persistent data in the browser frame, create an extension bar. Add an HTML file to your extension folder as the source for your extension bar and click New Bar, then choose the file from the pop-up menu. You are prompted to enter a label to identify your bar in the View menu. For more information, see "Adding Extension Bars" (page 35).
- Context Menu Items—Items your extension adds to contextual menus. Click New Context Menu Item to add an item. The interface expands to allow you to enter a title, identifier, and command for each context menu item. For more information, see "Adding Contextual Menu Items" (page 69).
- Toolbar Items—Buttons you are adding to the main Safari toolbar (*not* an extension bar). Click New Toolbar Item and you are prompted to enter a label, palette label, tooltip, image file, identifier, and command. The code that listens for the button click and executes the command goes in either your global HTML page or an extension bar. For details, see "Adding Buttons to the Main Safari Toolbar" (page 52).
- Menus—Pop-up menus you associate with a toolbar item you have created.
- Popovers—Pop-up windows containing HTML content that you associate with a toolbar item you have created.
- Start Scripts—Scripts to execute before a webpage is interpreted, usually a script that blocks unwanted content.
- End Scripts—Scripts to execute when the page load event occurs (roughly when a function specified in the body `onload` attribute would execute). Most scripts are End Scripts.

  For more information, see "Injecting Scripts" (page 76).
- Style Sheets—User style sheets to apply to browser content. For more information, see "Injecting Styles" (page 79).
- Whitelist—When injecting scripts or style sheets, only URLs specified in the whitelist are affected. If no whitelist is specified, all URLs are on the whitelist.
- Blacklist—When injecting scripts or style sheets, any URLs specified on the blacklist are skipped (scripts and style sheets are not injected).

  Add URLs to the whitelist or blacklist by clicking New URL Pattern.

  A URL pattern can include the `*` character to match any string. The `*` character can be used in any part of the `Domain` or `Path`, but not the `Scheme`.

  Examples:

  - `http://*/*`—matches all HTTP URLs.
  - `http://*.apple.com/*`—matches all webpages from apple.com.
  - `http://developer.apple.com/*`—matches all webpages from developer.apple.com.
  - `https://secure.A_BankForExample.com/accounts/*`—matches all webpages from the accounts directory of secure.A_BankForExample.com that are delivered over HTTPS.

    - `http://www.example.com/thepath/thepage.html`—matches one webpage.

  For more information, see "Access and Permissions" (page 95).

- Extension Settings—Persistent settings for your extension. Click New Settings Item to add a setting. Hidden settings are not displayed to the user. All other settings appear in a pane for your extension in Safari preferences. Each setting has a key (identifier), a type (such as a checkbox or text field), and an optional default value. For details, see "Settings and Local Storage" (page 99).

Building a Simple Extension

In order to create an extension, you need a minimum of two files:

1. A certificate (to obtain a certificate, join the Safari Developer Program at http://developer.apple.com).

   Install your certificate using Keychain Access (OS X) or the Extension Certificate Wizard (Windows). Double-clicking the certificate file launches the appropriate application.

2. A resource file, such as an HTML page, script, or CSS file. To create an extension bar, you need an HTML file.

   As the resource file for this example, create an HTML page that displays "Hello World" and save it as `helloworld.html`. An example is shown in Listing 2-1.

Listing 2-1 helloworld.html

```
<!DOCTYPE html>
<html>
<head>
<title>Hello World</title>
</head>
<body>Hello World!</body>
</html>
```

Follow these steps to create an extension bar that displays "Hello World".

1. Click the + button in Extension Builder and choose New Extension. Extension Builder creates an empty folder with the extension `.safariextension` and prompts you for a name and location. Name the new extension folder HelloWorld and have Extension Builder put it in your Documents folder. A generic icon is displayed with the name HelloWorld.

   This brings up the main Extension Builder interface, as shown in Figure 2-2 (page 27).

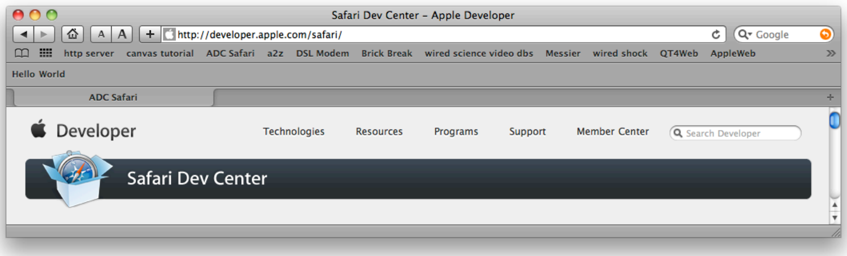
2. Put the `helloworld.html` file in the `HelloWorld.safariextension` folder.

   (You don't use Extension Builder for this step—just drag the file into the folder.)

3. Click New Bar in Extension Builder, enter a label such as `Hello World`, and choose `helloworld.html`from the pop-up menu. This tells Extension Builder to use `helloworld.html` as the source file for a new extension bar. The label appears in the View menu.

4. Click the Install button. Extension Builder creates a compressed folder with the extension `.safariextz`. This is your extension package. The package is installed, and a new extension bar is added to your browser window that displays "Hello World," as illustrated in Figure 2-4 (page 31).

Figure 2-4 Hello World



Notice that you can toggle the bar's visibility by name in the View menu, and disable or enable the extension itself in the Extensions pane of Safari preferences. You can edit and save `helloworld.html` to experiment with it. Click Reload in Extension Builder to build and install the modified version.

Accessing Resources Within Your Extension Folder

Your global HTML page, extension bars, injected scripts, and style sheets can all access resources within your extension folder, such as `.js` files, images, and other media. The resources must reside within your `.safariextension` folder when Extension Builder builds the extension (creates the compressed `.safariextz` package).

Using Relative URLs

Relative URLs are resolved differently for injected scripts and other extension resources.

Used from an injected script, relative URLs are relative to the webpage the script is injected into. From within an injected script—or any of its subresources, such as included scripts—resources in the extension folder can be loaded only by using an absolute URL (see "Using Absolute URLs" (page 33)).

For all other extension resources, including injected style sheets, relative URLs are relative to the source file within the extension folder.

This means you can access resources within the extension folder using relative URLs from the global page, extension bars, injected style sheets, or any of their subresources.

You can have nested folders within your extension folder. For example, your extension folder could contain a Scripts folder and an Images folder.

```
MyExtension.safariextz/
   Scripts/
      myScript.js
   Images/
      myImage.png
```

You can traverse the folder hierarchy with a relative URL by using `../` to go up a level. For example, from a script in the Scripts folder, you could load an image in the Images folder using this snippet:

`img.src='../Images/myImage.png'`.

Important: Do not begin a relative URL with a leading forward slash (/). Relative URLs must be relative to the file they are loaded from, not relative to the extension's folder.

This restriction applies in Safari 5.0.1 and later. If your extension uses relative URLs to access resources in your folder, and the extension was tested using Safari 5.0, you should retest it using Safari 5.0.1 or later to confirm that it does not use relative URLs that begin with the forward slash character.

Using Absolute URLs

You can use absolute URLs to access resources in your extension folder from any part of your extension.

Use of the `file:///` scheme is not allowed. Absolute URLs begin with `safari.extension.baseURI`, followed by the path within the folder and the filename. For example, using an absolute URL from a JavaScript function looks like this:

```
img.src = safari.extension.baseURI + 'Images/myImage.png'
```

Important: The base URI ends in a forward slash. Do not begin the path with another.

You must use JavaScript to obtain the absolute URL, as it changes each time Safari is launched. To load a resource from an HTML or CSS file using an absolute URL, you need to add some JavaScript to the source file.

Example: Loading a Background Image in CSS

An injected style sheet can add a background image to a website using images stored inside the extension. The easiest way to do this is to use a relative URL directly in CSS. For example:

```
body { background-image:url('../Images/paper.jpg'); }
```

To accomplish the same thing using an absolute URL, insert a few lines of JavaScript into your style sheet:

```
<script type = "text/javascript">
var myImage = safari.extension.baseURI + "Images/paper.jpg" ;
document.body.style.cssText = "background-image: url(" + myImage + ")";
</script>
```

Security

Use of the `file:///` scheme is not allowed.

The base URI returns a unique value each time Safari is run. It must be obtained at least once per session by your extension. It does not persist from session to session and it is not predictable.

This prevents outside scripts from determining the base URI of your extension and accessing its resources.

You can capture the base URI for your extension in a string and reuse the string within your code during a session, but you cannot store the string and reuse it from session to session.

Adding Extension Bars

Extension bars are toolbar-sized areas that serve as dedicated display space for extensions. Each extension bar is 30 pixels tall, so it's an appropriate place to put controls, a set of links, or a single line of information such as a scrolling headline or a stock ticker.

About Extension Bars

Extension bars are stacked between the bookmarks bar and the tab bar. Each extension bar's visibility can be toggled on and off using the View menu (OS X) or Action button (Windows). An extension can have multiple extension bars.

You designate an HTML file as the source of an extension bar using Extension Builder. For an example, see "Building a Simple Extension" (page 30).

Your extension bar can contain JavaScript functions defined in any of the usual ways, such as within the `head` element or in an included `.js` file.

Extension bar source files are loaded and interpreted each time a new browser window is opened. When the user opens a new window, Safari creates an instance of the `SafariExtensionBar` object for each bar. If no windows are open, there are zero extension bar instances. Hiding a bar using the View menu does not remove the instance. The extension's global HTML page and other extension bar files can access the extension bar and its properties using the `safari.extension.bars` array.

Multiple instances of an extension bar are independent, like the same webpage loaded in multiple windows. If an extension bar contains an audio player, for example, the play and pause button in a given extension bar act on the `audio` element in that bar, so if you start playing music, open a new window, and want to stop the music, you need to go back to the original window. To have the play and pause buttons in any copy of an extension bar act on the same `audio` element, put the `audio` element in a global HTML page.

The extension.bars Array

To address a particular instance of an extension bar, iterate through the `safari.extension.bars` array, using the `identifier` property of the extension bar to identify the particular bar, and the `browserWindow` property to identify the window instance. For example, to address the bar named Audio Controls in the active window, you might do this:

```
const bars = safari.extension.bars;
const activeBrowserWindow = safari.application.activeBrowserWindow;
for (var i = 0; i < bars.length; ++i) {
    var bar = bars[i];
    if (bar.browserWindow === activeBrowserWindow && bar.identifier === "Audio
Controls")
        {
        /* Do something. */
        }
}
```

Domain, URLs, and Access

An HTML file that acts as the source for an extension bar behaves pretty much the way any webpage would in a 30-pixel tall window, with the following exceptions:

- The domain of an extension bar is unique to the extension.
- Relative URLs are relative to the copy of the extension bar source file in the extension package. URLs outside of the extension package on the user's drive cannot be accessed.
- If you need to refer to a file in the package using an absolute URL, the URL is `safari.extension.baseURI + "relative-path/filename"`.
- An extension bar file has access to the Safari app's windows, tabs, contextual menu items, and toolbar items.
- An extension bar file can receive command events from contextual menu items or Safari toolbar items, as well as message events sent from injected scripts; this requires installing a listener function for the `command` event or `message` event.
- An extension bar file can send messages to the webpage proxy. These messages can be received by listener functions in injected scripts.
- An extension bar file has access to the global HTML page, if your extension has one.

- Extension bars can issue `XMLHttpRequest` to other domains. This allows your extension bar to read an RSS feed or a news site, for example, into a string that can be parsed for information to display in the bar. You set the domains your extension has permission to access using Extension Builder. For details, see "Access and Permissions" (page 95).

Displaying Content in an Extension Bar

The HTML in an extension bar file is automatically rendered in the extension bar. The shape of the extension bar, particularly its height, makes it suitable for particular tasks, such as the following:

- A toolbar for your extension.
- A specialized bookmarks bar.
- A ticker or news-crawl with headlines, stock price, weather, flight status, or other information suitable for single-line presentation that can be obtained using `XMLHttpRequest`.

The user can hide the extension bar using the View menu, but the extension bar page still loads every time a window is opened and any JavaScript still executes—only the display is suppressed by the View menu.

Creating an Extension Control Bar

To create an extension control bar, design a page that presents as a series of buttons, links, or controls, using the `title` attribute of your control elements to show tooltips. The shape, placement, and function of the controls are up to you, as long as they fit in the allotted space.

If the controls operate solely on the Safari app (manipulating windows and tabs, for example), or on elements declared in the extension bar, the code can be entirely within the extension bar file or the global HTML page.

If the extension bar triggers actions that need access to content loaded in a browser tab, however, the code that acts on the content must be injected into the webpage. The extension bar cannot call functions in injected scripts directly. The extension must send a message to the webpage proxy, and the injected script must have a listener function registered for the `"message"` event. See "Interacting with Injected Scripts" (page 41).

The following example, Listing 4-1, is HTML that displays two buttons. Setting the `title` attribute on the buttons creates tooltips. The `<audio>` element loads an audio file from a remote server. The buttons play and pause the music.

Listing 4-1 Music player bar

```
<!DOCTYPE html>
<html>
```

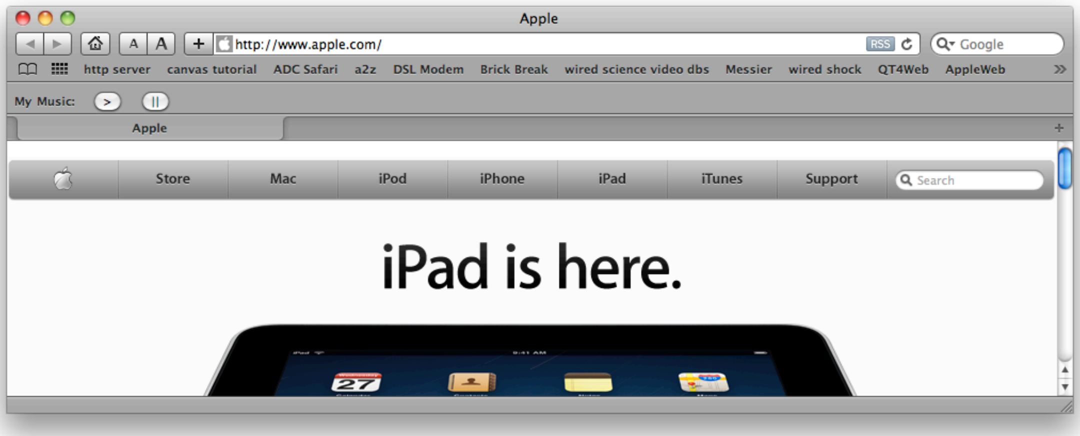
```
<head>
    <title>Music Player Extension Bar</title>
    <script type="text/javascript">
    function playIt() {
       document.getElementById("music").play();
        }
    function pauseIt() {
       document.getElementById("music").pause();
        }
    </script>
</head>
<body>
My Music:
&nbsp;&nbsp;
<input type=button value=">" onclick="playIt()"  title="Play">
&nbsp;&nbsp;
<input type=button value="||" onclick="pauseIt()" title="Pause">
<audio id="music" src="http://homepage.mac.com/qt4web/testmusic.m4a">
</audio>
</body>
</html>
```

To make this example into an extension bar, follow these steps:

1. Save the example as an HTML file.
2. Open Extension Builder, click +, choose New Extension, and give the extension a name (see "Using Extension Builder" (page 25)).
3. Drag the HTML file into the extension folder you just created.
4. Click New Bar in Extension Builder and choose the HTML file from the pop-up menu.

Click Install. You should see the music player toolbar in Safari, as shown in Figure 4-1.

Figure 4-1 Music player toolbar



Note that you can create several instances of the music bar by opening new windows, and that each bar plays and pauses independently.

Working with Windows and Tabs

An extension bar can use either HTML or JavaScript to display content in a tab.

Note: Be sure to set your extension's website access to Some or All in Extension Builder before working with tabs—most tab properties return `undefined` unless your extension has access to the domain of the URL loaded in the tab.

A link in an extension bar, such as `<a src=URL> link text </a>`, opens the linked URL in the active browser tab, just as it would from a webpage. Unlike a normal webpage, the extension bar is not replaced with the linked file, however. Consequently, an extension bar can contain a set of persistent links, similar to the bookmarks bar.

The standard `window.open()` method cannot be used to open a new tab and window from an extension bar. Instead, extension bars have access to the `SafariApplication`, `SafariBrowserWindow`, and `SafariBrowserTab` classes, which allow you to open, close, activate, and manipulate windows and tabs.

For example, this opens a window and returns the active tab:

```
var newTab = safari.application.openBrowserWindow().activeTab;
```

And this opens a new tab in the window containing the extension bar:

```
var newTab = safari.self.browserWindow.openTab();
```

For more details, see "The Windows and Tabs API" (page 81).

Listing 4-2 (page 40) shows how to implement an extension bar that opens content in a new tab of the extension bar's window.

Listing 4-2 Safari developer reference bar

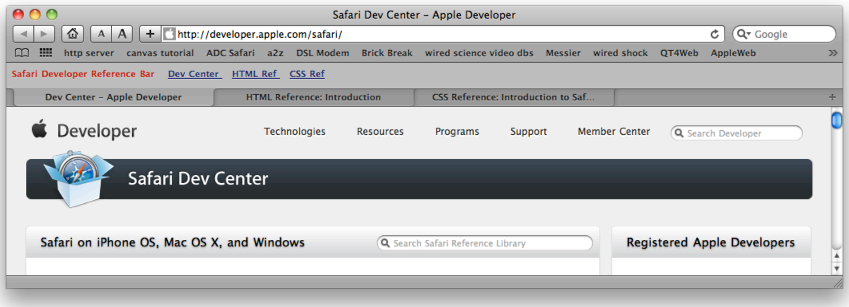
```
<!DOCTYPE HTML>
<html>
<head>
    <title>Safari Developer Reference</title>

    <script type="text/javascript">
    var server="http://developer.apple.com/";
    var reflib="safari/library/documentation/AppleApplications/Reference/"
    function openInTab(source){
        var newTab=safari.self.browserWindow.openTab();
        newTab.url=source;
        }
    </script>

</head>
<body style="color:#C02020;background:#C0C0C0;">
Safari Developer Reference Bar
&nbsp;&nbsp;
<a href="javascript:openInTab(server+'safari/');"> Dev Center </a>
&nbsp;&nbsp;
<a href="javascript:openInTab(server+reflib+'SafariHTMLRef/');"> HTML Ref </a>
&nbsp;&nbsp;
<a href="javascript:openInTab(server+reflib+'SafariCSSRef/');"> CSS Ref </a>
</body>
</html>
```

Figure 4-2 shows what the example looks like when all three tabs have been opened.

Figure 4-2 Reference extension bar


Safari Dev Center – Apple Developer
http://developer.apple.com/safari/
Google
http server
canvas tutorial
ADC Safari
a2z
DSL Modem
Brick Break
wired science video dbs
Messier
wired shock
QT4Web
AppleWeb
Safari Developer Reference Bar
Dev Center
HTML Ref
CSS Ref
Dev Center – Apple Developer
HTML Reference: Introduction
CSS Reference: Introduction to Saf...
Developer
Technologies
Resources
Programs
Support
Member Center
Search Developer
Safari Dev Center
Safari on iPhone OS, Mac OS X, and Windows
Search Safari Reference Library
Registered Apple Developers

Interacting with Injected Scripts

Extension bars cannot address the content of webpages, but they can interact with injected scripts indirectly, by sending and receiving messages. There are two primary reasons to do this:

1. You might activate or control an injected script using controls in an extension bar.
2. Your extension bar might contain code that your injected script needs to call.

Injected scripts are interpreted each time the user loads a URL the script applies to, including subframes, so it's important to keep your injected scripts lightweight. Otherwise, the load time for every page is slowed. An extension bar is loaded only once per window, regardless of how many tabs are opened and URLs are loaded. Consequently, if a script needs user controls, it's better to put them in an extension bar than to inject them into each page.

Similarly, if your script needs to perform significant calculations, or refer to a large table of data, it's better to load the data or large block of code once per window than once per page. In general, it's better still to load the code or data in your global HTML page and do it only once per session, but in cases where you need one copy per window, the extension bar can be used.

In order to control an injected script, your extension bar needs to send a message by calling the `SafariWebPageProxy` object's `dispatchMessage()` method. The proxy stands in for the web content, which can be accessed as the `page` property of a `SafariBrowserTab` object, which is in the `tabs` array or `activeTab` property of a `SafariBrowserWindow` object, so sending a message to a script takes the general form:

```
safari.application.activeBrowserWindow.activeTab.page.dispatchMessage(msgName, data)
```

or:

```
safari.application.browserWindows[n].tabs[n].page.dispatchMessage(name, data)
```

The injected script in the specified page must have a listener function registered for `message` events in the `SafariContentWebPage` object (`safari.self`):

```
safari.self.addEventListener("message", respondToMessage, false);
```

In order to execute functions in your extension bar in response to a request from an injected script, you must define and register a listener function for `message` events in your extension bar. You should generally register your listener function at the window level. For example:

```
safari.self.browserWindow.addEventListener("message", respondToMessage, false);
```

For more details and examples, see "Messages and Proxies" (page 87).

Message-Passing Example

The following example shows an extension bar file, Listing 4-3 (page 42), and an injected End Script, Listing 4-4 (page 43). The extension bar has a button to send a message and a text field that changes when it receives a message. The script adds a text field to webpages that changes when it receives a message.

To see the example in action, follow these steps:

1. Copy the text in the following listings into a text editor and save as `extensionbar.html` and `injected.js`.
2. Create an extension folder using Extension Builder and drag `injected.js` and `extensionbar.html` into the folder.
3. In Extension Builder, set Extension Website Access to All.
4. Click New Bar in Extension Builder and choose `extensionbar.html`.
5. Click New Script under Injected Extension Content—End Scripts: and choose `injected.js`.
6. Click Install, then load a webpage in Safari to load the script.

Note: You must designate `injected.js` as an End Script. If you attempt to run it as a Start Script, `document.body` is undefined, as the webpage that the script is being injected into is not yet parsed.

Listing 4-3 Extensionbar.html

```
<!DOCTYPE HTML>
<html>
<head>
```

```
<script type="text/javascript">


    function sendMessage() {
        document.getElementById("textField").innerHTML="Sending message...";
       safari.application.activeBrowserWindow.activeTab.page.dispatchMessage("hey",
 "there");
    }


    function respondToMessage(messageEvent) {
        if(messageEvent.name === "gotIt")
        document.getElementById("textField").innerHTML=messageEvent.message;
    }


    safari.self.browserWindow.addEventListener("message",respondToMessage,false);


</script>
</head>
<body> Message Sender Bar &nbsp;&nbsp;
    <input type="button" value="Send" onclick="sendMessage()" >
    <span id="textField">...waiting... </span>
</body>
</html>
```

Listing 4-4 Injected.js

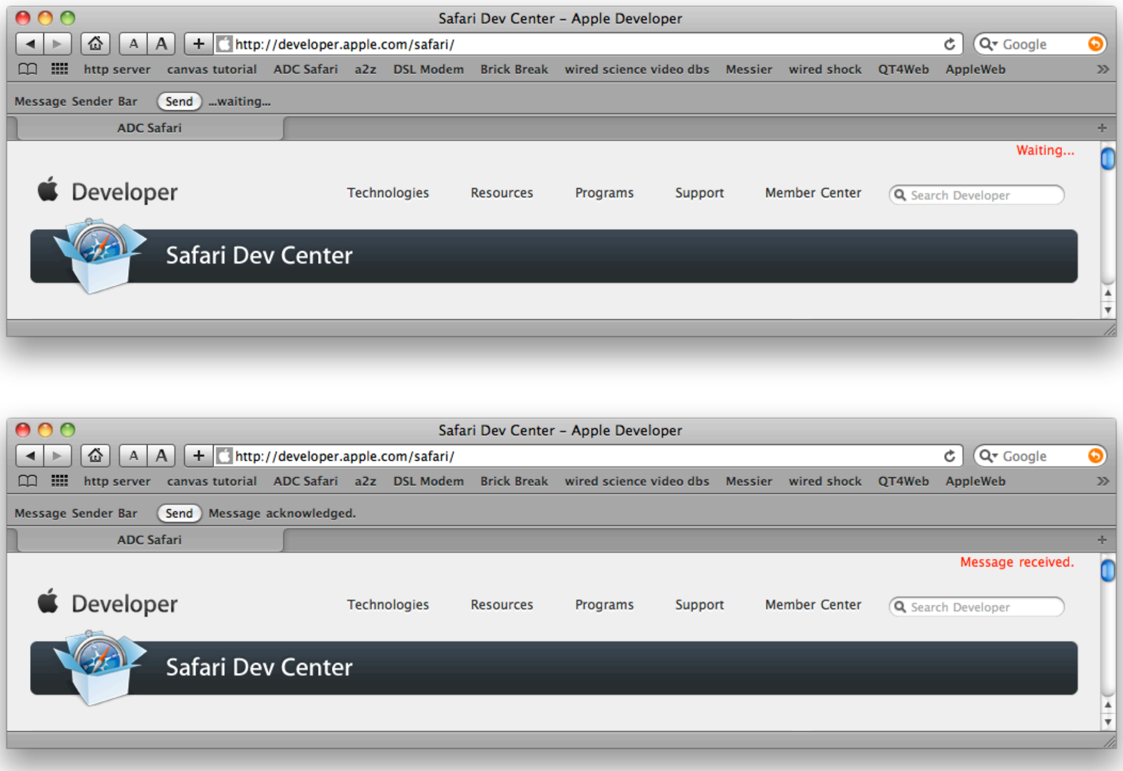
```
var theBody = document.body;
// create a para and insert it at the top of the body
var element = document.createElement("p");
element.id = "status";
element.style.cssText = "float:right; color:red";
element.textContent = "Waiting...";
theBody.insertBefore(element, theBody.firstChild);


function replyToMessage(aMessageEvent) {
   if (aMessageEvent.name === "hey") {
    document.getElementById("status").textContent="Message received.";
```

```
        safari.self.tab.dispatchMessage("gotIt","Message acknowledged.");
    }
}
// register for message events
safari.self.addEventListener("message", replyToMessage, false);
```

Figure 4-3 shows the extension bar and the webpage modified by the injected script, before and after sending messages.

Figure 4-3 Before and after



...waiting...

Waiting...

Message acknowledged.

Message received.

Adding a Global HTML Page

A global HTML page is a place for you to put JavaScript code, data tables, and other resources requiring no user interface that your extension needs to load only once per Safari session. The global page is not mandatory. You can have at most one global page per extension.

The main uses for a global page are:

- To hold the logic for handling Safari toolbar items.
- To hold the logic for handling extension menus and contextual menu items.
- To hold the logic for handling settings changes.
- To hold large blocks of logic or data used by extension bars.
- To hold large blocks of logic or data used by injected scripts.

You can also use the global page to hold logic or data used by a popover, but if the popover is specified in Extension Builder, the popover file loads only once, just like the global page file, so there is no gain in efficiency. If you have multiple popovers that use the same data or logic, however, it makes sense to have one copy in the global page, instead of a copy in each popover.

Adding a Global Page in Extension Builder

First create the global HTML page. The global HTML page is an HTML file that is loaded but never displayed. It can contain JavaScript functions declared in `script` elements or in external `.js` files included using the `script` element's `src` attribute. The `.js` files can be located inside the extension folder and referenced by relative URL.

If you have not already done so, click + in Extension Builder and choose New Extension. You are prompted to give the extension a name and choose a location for it. Extension Builder creates a folder with the name you choose and the file extension `.safariextension`.

Drag the HTML file that you want to use as your global page into the extension folder using the Finder or Windows file system. Drag in any external `.js` files or other resources the global page needs as well.

Click Global Page File in Extension Builder and choose the file from the pop-up menu.

Note: When you add a global page, an Inspect Global Page button is added to Extension Builder. Click this button to invoke Web Inspector to help debug your global page. For details, see "Debugging Extensions" (page 106).

Handling Toolbar Items

The global page is the ideal place for the logic that handles toolbar items. You can also put the logic in an extension bar, but since the logic doesn't need to be loaded on a per-window basis and doesn't need any display space, it usually belongs in a global page.

To handle toolbar items, add a listener function for `"command"` events in your global page. Have the listener function test the event name to see if it is the name of the command you specified for the toolbar item in Extension Builder. If the event name is your command name, the user has clicked your toolbar item and you should execute the command.

If there is any possibility that the toolbar item should be disabled, add a listener function for the `"validate"` event as well. If the event name is the same as your command name, determine whether the toolbar item should be disabled. If so, set `event.target.disabled = true`.

For details and examples, see "Adding Buttons to the Main Safari Toolbar" (page 52).

In Safari 5.1 and later, a toolbar item can also generate `"menu"` or `"popover"` events. See "Adding Extension Menus" (page 59) and "Adding Popovers" (page 64) for details.

Handling Contextual Menu Items

The global page is also the ideal place for the logic that handles contextual menu items. Again, you can put the logic in an extension bar, but since the logic doesn't need to be loaded on a per-window basis and it doesn't need any display space, it usually belongs in a global page.

To handle contextual menu items, add a listener function for `"command"` events in your global page. Have the listener function test the event name to see if it is the name of the command you specified for the contextual menu item in Extension Builder. If the event name is your command name, the user has chosen your contextual menu item and you should execute the command.

If there is any possibility that the contextual menu item should not be displayed, add a listener function for the `"validate"` event as well. If the event name is the same as your command name, determine whether the contextual menu item should be displayed. If not, set `event.target.disabled = true`.

If you need to obtain information from the webpage, such as the element that was clicked to initiate the contextual menu, you can add event listeners for the `"contextmenu"` event in the global page and an injected script. The script's listener is called first, and can set pass information to the global page.

For details and examples, see "Adding Contextual Menu Items" (page 69).

Support Logic for Extension Bars and Popovers

Functions and global variables declared in the global page can be accessed directly from extension bars or popovers, as properties of `safari.extension.globalPage.contentWindow`. So, for example, if you define a `myCalc()` function in your global page, you can call it from an extension bar or popover using `safari.extension.globalPage.contentWindow.myCalc()`.

To simplify things, declare a constant in your extension bar or popover, such as:

```
const myGlobal = safari.extension.globalPage.contentWindow;
```

You can then access the `myCalc()` function from your extension bar or popover using `myGlobal.myCalc()`.

The global page has access to the same API as an extension bar or popover, so it can do all the same things, except that it has no visible display area of its own. Consequently, it's easy to move large chunks of code from an extension bar or popover to your global page. Listing 5-1 shows how to move logic from an extension bar, but it would work the same way from a popover.

This is an example of an extension bar that has a button. Clicking the button performs a simple calculation.

Listing 5-1 Moving logic to your global page

```
<!DOCTYPE HTML>
<html>
<head>
   <title>old extension bar page</title>
   <script type="text/javascript">

   var theAnswer = 0;
    function calcThis(x) {
       x++;
       theAnswer = x;
    }
```

```
     function doButton() {
        calcThis(theAnswer);
        var mButton = document.getElementById("myButton");
        mButton.value = ("Increment " + theAnswer);
     }
   </script>
</head>
<body>
   <input type="button" value="Increment 0"
    onclick="doButton();" id="myButton" >
</body>
</html>
```

Here's a version of the same extension bar, with the calculation moved to the global page. The code exported to the global file is unchanged. A constant is defined in the extension bar and prepended to anything called in the global file.

```
<!DOCTYPE HTML>
<html>
<head>
   <title>global page</title>
   <script type="text/javascript">

    var theAnswer = 0;
    function calcThis(x) {
      x++;
      theAnswer = x;
    }

   </script>
</head>
<body> </body>
</html>

<!DOCTYPE HTML>
```

```
<html>
<head>
   <title>extension bar page</title>
   <script type="text/javascript">

   const myGlobal = safari.extension.globalPage.contentWindow;

   function doButton() {
      myGlobal.calcThis(myGlobal.theAnswer);
      var mButton = document.getElementById("myButton");
      mButton.value = ("Increment " + myGlobal.theAnswer);
   }

   </script>
</head>
<body>
   <input type="button" value="Increment 0"
    onclick="doButton();" id="myButton" >
</body>
</html>
```

You can also make function calls from the global page into an extension bar or popover. Calling functions in this direction is slightly more complex. You still call into the content window, but there is an array of extension bars and an array of popovers, so you need to specify which element of the array you are addressing, even if your extension has only one bar or popover. There is also an instance of each extension bar and all its functions per open window. Your global page needs to identify which instance it wants to access, or iterate through them to access them all.

For example, suppose your extension has a single extension bar; the following snippet iterates through the extension bar instances and calls the `doSomething()` function defined in each one, but calls the `doSomethingSpecial()` function only on the extension bar in the active window:

```
const myBars = safari.extension.bars;
function updateAllBars {
    for (var i = 0; i < myBars.length; ++i) {
        var barWindow = myBars[i].contentWindow;
        barWindow.doSomething();
        var myWindow = safari.application.activeBrowserWindow;
        if (myBars[i].browserWindow == myWindow) {
            barWindow.doSomethingSpecial();
        }
    }
}
```

To call a function in a popover, you need to identify the particular element in the `popovers` array that you want to call into, but there is a single popover instance shared by all windows. If your extension has a single popover, you can call a function in the popover by calling `safari.extension.popovers[0].contentWindow.functionName()`.

Support Logic for Injected Scripts

JavaScript functions and variables in your global page cannot be called directly from injected scripts, but injected scripts can send messages that trigger functions in the global page, and the global page can send messages to injected scripts that trigger functions or contain the result of calculations.

For details and examples, see "Messages and Proxies" (page 87).

Working with Windows and Tabs

Note: Be sure to set your extension's website access to Some or All in Extension Builder before working with tabs—most tab properties return `undefined` unless your extension has access to the domain of the URL loaded in the tab.

The global HTML page itself is not displayed, but it can open windows and tabs and use them to display content. The standard `window.open()` method cannot be used to open a new tab and window from the global HTML page, however. Instead, the global page has access to the `SafariApplication`, `SafariBrowserWindow`, and `SafariBrowserTab` classes, which allow you to open, close, activate, and manipulate windows and tabs.

For example, this opens a window and returns the active tab:

```
var newTab = safari.application.openBrowserWindow().activeTab;
```

In Safari 5.1 and later, there are events generated when a window or tab opens, closes, gains or loses focus, or when the user navigates to another page.

For details, see "The Windows and Tabs API" (page 81).

Adding Buttons to the Main Safari Toolbar

Safari has a user-customizable toolbar that can contain a selection of buttons, such as a Home button, Zoom button, and New Tab button. Your extension can define new toolbar items that can be installed in the toolbar. These items also appear in the Customize Toolbar panel.

You control the actions of a toolbar item from either the global HTML page or from an extension bar by installing a listener function for the `"command"` event.

Your extension can add more than one button to the toolbar, but if you are adding more than a few, you should not have them installed by default; you might also consider creating an extension bar for them instead.

Adding a button requires three steps: creating an image, filling out the appropriate fields in Extension Builder, and adding logic to make the button do something.

In Safari 5.1 and later, you can add either a pop-up menu or a popover window to a toolbar item. See "Adding Extension Menus" (page 59) and "Adding Popovers" (page 64) for details.

Note: You can assign either a menu *or* a popover to a toolbar item, but not both.

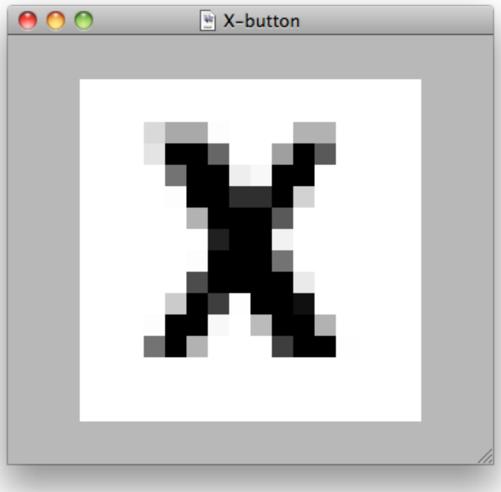
Creating an Image

Buttons on the Safari toolbar are largely transparent, allowing them to be filled with the appropriate gradient for the current OS X or Windows user interface. You do not need to draw the button itself, only the opaque part of its contents.

If you are used to working with alpha channels, create a 16x16 image consisting solely of an 8-bit alpha channel, with alpha set to 255 for the transparent part of the button (including the outline); alpha set to 0 for the opaque parts of the button, which will appear in black; and intermediate alpha values for anti-aliasing.

If you are not used to working with alpha channels, create a 16x16 pixel image with a transparent background. Fill in the button contents in black, or simply set the opacity for those pixels to 100%. If you draw or put text on the button in black, notice that some pixels are gray. This is anti-aliasing. These pixels are not completely black, so they are shaded. This is illustrated in Figure 6-1.

Figure 6-1 Anti-aliased text in black on transparent background



If you save your image as-is, these shaded pixels appear as black because they are completely opaque. To preserve the anti-aliasing, make these gray pixels partly transparent. Depending on your image editor, this may be accomplished by partly erasing the pixels or by setting the pixel opacity directly.

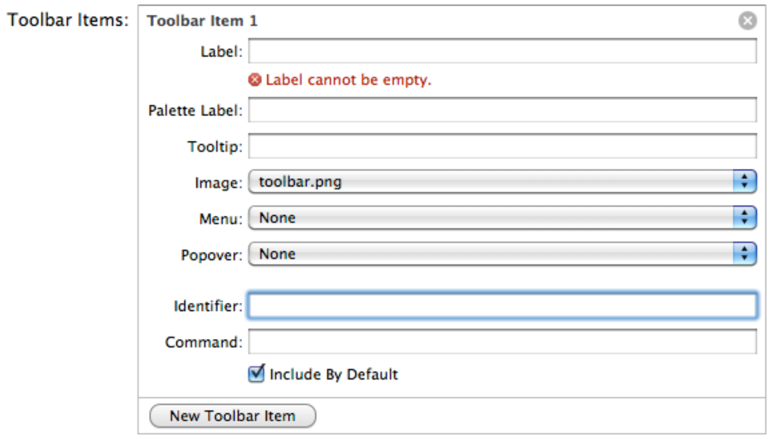
Save your image as a `.png` file.

Setting Up Extension Builder

If you have not already done so, click the + button in Extension Builder, choose New Extension, and give your extension a name. Create an image for each button you are adding and drag it into your extension folder.

Click New Toolbar Item in Extension Builder. This expands the Toolbar Items pane, as shown in Figure 6-2.

Figure 6-2 Adding toolbar items


Toolbar Items: Toolbar Item 1
Label:
Label cannot be empty.
Palette Label:
Tooltip:
Image: toolbar.png
Menu: None
Popover: None
Identifier:
Command:
[✓] Include By Default
New Toolbar Item

- Enter a label for your toolbar item. This is a text label that is displayed if the bar has more buttons than can be shown and the user clicks the chevron to see the overflow.
- Enter a palette label for your toolbar item. This is the text label that is displayed when the user is customizing the toolbar. It can be the same as the label, or it can be a bit longer if that makes the function of your toolbar item clearer. If this field is left blank, the label is used.
- Enter a tooltip for your toolbar item. This is the text displayed when the mouse pointer hovers over the item. If this field is left blank, the label is used.
- Drag a `.png` image file into your extension folder. The image should ideally be 14x14 or 16x16 pixels. Larger images are cropped at 18x18 pixels. The image is an alpha mask. It must contain an 8-bit alpha channel defining the part of the button face that is drawn, and nothing else. Using an alpha mask allows your button to blend in with Safari's native toolbar buttons on different platforms, even if the Safari UI should change. Once you have an image file in your extension folder, you can choose it from the Image pop-up menu. An image is required.

  You can include an alternate image at double the resolution for use on high-resolution displays. Use the same name for the alternate image file, but append the characters `@2x`. For example, if your image is named `myToolBarImg.png`, the high-resolution image should be named `myToolBarImg@2x.png`.
- If you have created a menu to be triggered by this toolbar item, select it from the pop-up menu.
- If you have created a popover to be triggered by this toolbar item, select it from the pop-up menu.
- Enter an identifier. This field is required. The identifier is your name for the toolbar item. The name must be unique in your extension. You can identify the toolbar item later by iterating through the array of items and checking the value of the `identifier` property:

```
var itemArray = safari.extension.toolbarItems;
for (var i = 0; i < itemArray.length; ++i) {
    var item = itemArray[i];
    if (item.identifier == "my lovely button")
       {
       /* Do something. */
       }
```

- Enter a command name. This is the `event.command` property of the event that is generated when the user clicks your item in the toolbar. It does not need to be unique. For example, you might have both a toolbar button and a contextual menu item that issue the same command. If this field is left blank, the identifier is used.

  If you add a menu or a popover to the toolbar item, you may want to leave the Command field blank. Leaving the field blank sets the `command` property to `null` and causes the menu or popover to be displayed immediately when the user clicks your toolbar item. Otherwise, the command executes when the button is clicked, and the menu or popover is displayed only when the user clicks and holds.

If you select the Include By Default checkbox, the item is installed in the toolbar when the extension is installed. Otherwise, the user must choose to add the item in the Customize Toolbar window.

Responding to Commands

When the user clicks the button, Safari emits a `"command"` event, if one is specified. The `command` property of the event is the string you entered in the Command field in Extension Builder. If you left the Command field blank, the identifier is used instead.

Note: In Safari 5.1 and later, you can assign a popover or an extension menu to a toolbar item. If you leave the Command field blank, the menu or popover is displayed and no `"command"` event is generated.

You can respond to the `"command"` event by installing a listener function in your global HTML page or an extension bar or popover.

You can't receive the `"command"` event in an injected script. If you need the command to initiate an action in an injected script, respond to the command in the global HTML page or an extension bar and send a message to the script. For details, see "Messages and Proxies" (page 87).

At various times, such as when a tab is added, Safari will ask you to validate the command. If there is any possibility that the command could be invalid, you should add a listener function for the `"validate"` event.

Your validate function should verify that the command is ready and should be enabled. For example, if your command reloads the active tab, you should verify that the active tab has a URL to reload. If the tab is empty, your validate function should disable the button. You can also modify the item's image to reflect a modified behavior.

The `"validate"` event is fired when an item is added to the toolbar, so this is also a good moment to update badges.

You can have multiple UI items that issue the same command, such as a toolbar item and a contextual menu item. You can use the same event handlers, regardless of the source.

If your functions are part of the global HTML page, you should register your listener functions with the app:

```
safari.application.addEventListener("command", myCommandHandler, false);

safari.application.addEventListener("validate", myValidateHandler, false);
```

If your functions are part of an extension bar, you should register your listener functions with the extension bar's parent window:

```
safari.self.browserWindow.addEventListener("command", myCommandHandler, false);

safari.self.browserWindow.addEventListener("validate", myValidateHandler, false);
```

While you can implement the event handlers in either a global HTML page or in an extension bar, it is more efficient to use a global HTML page, because the code is loaded only once, when Safari loads the extension, instead of once per window.

Deciding Where to Respond

When your button is clicked, a `"command"` event is generated. You can listen for the event in either a global HTML page, a popover, or in an extension bar. If you put the event handler in the global file or in a popover, you should register for the event at the app level. If your event handler is in an extension bar, you should register with the extension bar's parent window.

The difference is that there is only one instance of the popover or global HTML page's functions, but there is an instance of an extension bar in every open window.

If every instance of the extension bar registers for events at the app level, every instance responds to the command. If each instance registers with its parent window, only the instance in the window where the button is clicked responds to the command.

Using the global page is more efficient, as it loads only once. Furthermore, you shouldn't create an empty extension bar just to hold an event handler. If your extension has a bar, however, it might make sense to put the event handler there, particularly if the action it takes is window-specific, like rearranging the tabs.

The only time you might want to put an event handler in the extension bar and register it with the app is if your command acts on all open windows. Then your choice would be to iterate through the windows in a global function or have a function local to each window that acts independently.

If You Respond from a Global HTML Page

- Register with the app: `safari.application.addEventListener()`.
- The window the event comes from is: `event.target.browserWindow`.

If You Respond from an Extension Bar

- Register with the parent page: `safari.self.browserWindow.addEventListener()`.
- The window the event comes from is: `self.browserWindow` or `event.target.browserWindow`. The two are equivalent.

Example: Implementing a Reload Button

Listing 6-1 (page 57) responds to the `"reload-page"` command event and the `"reload-page"` validate event. The validate handler disables the control if there's nothing for it to do, but the command handler checks anyway, in case things have changed since validation.

Listing 6-1 Reload command and validate handlers

```
function performCommand(event)
{
    if (event.command === "reload-page") {
        var currentURL = event.target.browserWindow.activeTab.url;
        if (currentURL)
            event.target.browserWindow.activeTab.url = currentURL;
    }
}

function validateCommand(event)
{
    if (event.command === "reload-page") {
        // Disable the button if there is no URL loaded in the tab.
        event.target.disabled = !event.target.browserWindow.activeTab.url;
    }
}

// if event handlers are in the global HTML page,
// register with application:
safari.application.addEventListener("command", performCommand, false);
safari.application.addEventListener("validate", validateCommand, false);
// if event handlers are in an extension bar,
// register with parent window:
// safari.self.browserWindow.addEventListener("command", performCommand, false);
// safari.self.browserWindow.addEventListener("validate", validateCommand, false);
```

Adding Extension Menus

In Safari 5.1 and later, you can create menus that pop up when the user clicks a toolbar item that you have created. If the toolbar item's `command` property is set to `null`, clicking the toolbar item displays the menu immediately. If the `command` property is *not* `null`, the user must press and hold in order to see the menu—clicking alone sends the command instead. You can set a toolbar item's `command` property to `null` either programmatically or simply by leaving the toolbar item's Command field blank in Extension Builder.

You normally associate a menu with a toolbar item using Extension Builder, but you can also attach a menu to a toolbar item programmatically by setting the toolbar item's `menu` property.

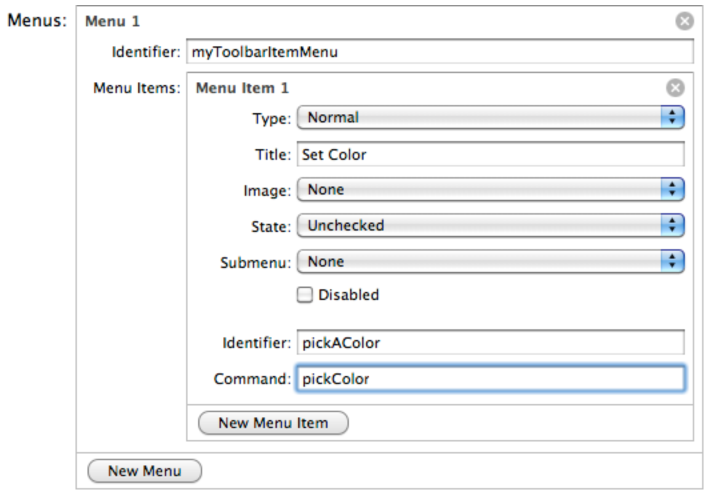
When the user clicks your toolbar item, a `"menu"` event is generated prior to displaying the menu, followed by `"validate"` events for each menu item. When the user chooses one of your menu items, a `"command"` event is generated for that menu item.

Note: You can assign either a menu or a popover to a toolbar item, but not both.

Setting Up Menus in Extension Builder

To add a menu to a toolbar item, start by clicking the New Menu button in Extension Builder; the Menus pane expands, as shown in Figure 7-1.

Figure 7-1 Extension Menus pane



Menus: Menu 1
Identifier: myToolbarItemMenu
Menu Items: Menu Item 1
Type: Normal
Title: Set Color
Image: None
State: Unchecked
Submenu: None
Disabled
Identifier: pickAColor
Command: pickColor

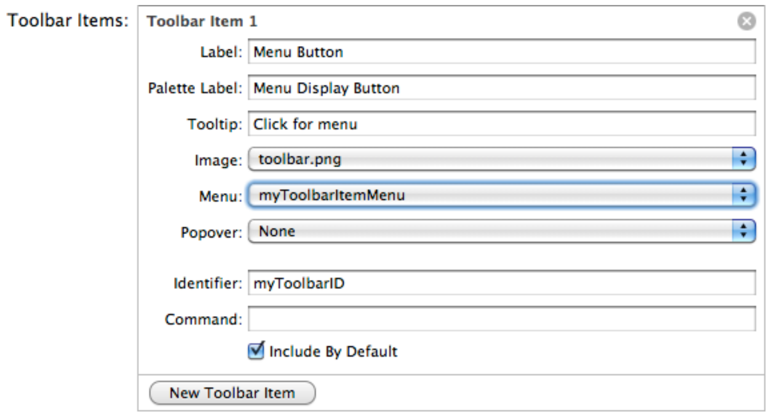
Give the menu a unique identifier, then add any menu items that normally appear in your menu—you can add and delete menu items at runtime as well.

For each menu item, set the following values:

- Type—choose Normal for a menu item, or Separator for a line separating groups of menu items.
- Title—the text shown on your menu item.
- Image—the filename of an image you created for this menu item. The image must be in your extension folder for you to see it in the pop-up menu. This value is optional. You can also assign the URL of an image file to the menu item dynamically at runtime.
- State—the default menu item state: checked, unchecked, or mixed. You can modify the state at runtime.
- Submenu—the menu to activate if this menu item is selected. Create submenus by clicking New Menu.
- Disabled—select this option if the menu item should be grayed-out by default. You can change this property at runtime.
- Identifier—a unique identifier for this menu item.
- Command—the `command` property of the `"command"` event generated when this menu item is clicked. If this field is left blank, the identifier is used as the command.

After you have created your menu, go to the toolbar item you want to associate with this menu and click the Menu button. Then choose this menu's identifier from the pop-up menu, as illustrated in Figure 7-2.

Figure 7-2 Attaching a menu to a toolbar item


Toolbar Items: Toolbar Item 1

Label: Menu Button
Palette Label: Menu Display Button
Tooltip: Click for menu
Image: toolbar.png
Menu: myToolbarItemMenu
Popover: None
Identifier: myToolbarID
Command:
[✓] Include By Default

Creating an Image (Optional)

You can create an image to display alongside any item in your menu. For best results, create your image as a 16x16 pixel image on a transparent background. Images larger than 16x16 are scaled down. Unlike the image in a toolbar item, colored images are displayed normally.

Save your image as a `.png` file. Put the image file inside your extension folder. Use a filename that is URL-friendly (no blank spaces, ampersands, or periods, for example).

Choose the filename from the pop-up menu in the Image field for your menu item.

You can include an alternate image at double the resolution for use on high-resolution displays. Use the same name for the alternate image file, but append the characters `@2x`. For example, if your image is named `myToolBarImg.png`, the high-resolution image should be named `myToolBarImg@2x.png`.

You can assign an image to a menu item at runtime, in which case you supply the URL of the image. The URL is absolute and can refer to either an image file in your extension package or an image on the web.

Responding to Events

In your global HTML file, register on the Safari app for `"command"` events to receive commands when one of your menu items is clicked. You may also want to add listener functions for `"validate"` and `"menu"` events if your menu changes dynamically. For example:

```
safari.application.addEventListener("validate", validateHandler, true);

safari.application.addEventListener("menu", menuHandler, true);

safari.application.addEventListener("command", commandHandler, true);
```

Safari generates `"validate"` events on the toolbar item periodically, including just before your menu is displayed. Use this event if you need to dynamically enable and disable the toolbar item itself, or to update a badge on the toolbar item to reflect changes in your menu. If your toolbar item is always enabled and has no badge, you don't need to respond to this event. Check to see whether the `event.target.identifier` property for the `"validate"` event is the identifier for your toolbar item. For example:

```
if (event.target.identifier !== "myToolbarItemID") return;
```

Safari generates a `"menu"` event when your menu is about to be displayed. This event is your opportunity to add, remove, or modify menu items dynamically. If your menu does not change dynamically, you don't need to respond to this event. The event target is a menu, so the `event.target.identifier` property for the `"menu"` event is the identifier of the target menu.

Safari generates a `"validate"` event for each menu item prior to displaying your menu, but after the `"menu"` event. You can use this event to dynamically enable, disable, or modify individual menu items. If your menu items do not change dynamically, you do not need to respond to these events. The `event.target.identifier` property for the `"validate"` event is the identifier of your menu item.

When the user clicks one of your menu items, Safari generates a `"command"` event. The `event.command` property is the command value assigned to your menu item. You need to listen for and respond to this event in order for your menu to do anything. You typically write a single command handler for your menu, with a case statement to sort the commands by menu item.

Your global HTML page is the best place to put event handlers.

If your menu changes depending on things such as currently open windows or tabs, you may want to listen for other events as well. See "The Windows and Tabs API" (page 81) for details.

Modifying the Menu Dynamically

You can dynamically add, delete, and modify menu items at runtime. You can also create a menu at runtime and attach it to a toolbar item you have created. There is a `menus` array that is a property of the `extension` object.

Important: You cannot add, delete, or modify menus or menu items while the menu is visible. Attempting to do so fails and causes an exception to be thrown. Check the `visible` property of your menu instance before making any changes, and hide the menu by calling `myMenu.hide()` if you must modify or delete it immediately.

- Create a menu by calling `myMenu = safari.extension.createMenu(ID)`.
- Assign a menu to a toolbar item by setting the toolbar item's `menu` property to the menu.

  You can obtain an instance of your toolbar item by iterating through the `extension` object's `toolbarItems` array and testing the `identifier` property for your toolbar item.
- Delete a menu by calling `safari.extension.removeMenu(ID)`. Set the toolbar item's `menu` property to `null` before deleting its menu.
- Delete a menu item by calling `removeMenuItem(index)` on the `menu` object.
- Add a menu item by calling `appendMenuItem(identifier, title)` or `insertMenuItem(index, identifier, title)` on the `menu` object.
- Add an image to a menu item by setting the menu item's `image` property to the URL of an image file (this can be the filename of an image in your extension folder or the HTTP or HTTPS URL of an image on the Internet).

- Modify a menu item by changing its `title`, `disabled`, `checkedState`, and `image` properties. See `SafariExtensionMenuItem` Reference for details.

  Note: Disabled menu items in extension menus are displayed as grayed-out, unlike contextual menu items, which are not displayed when they are disabled.

- Add a separator to your menu by calling `appendSeparator()` or `insertSeparator(index)` on the menu object.

Adding Popovers

A popover is an HTML window that pops up over the existing windows—no navigation required. The popover window closes automatically when the user changes focus (by clicking in another window, for example). Use popovers to display intermediate amounts of data—more than fits conveniently in an extension bar, but less than you would need a full page to display.

In Safari 5.1 and later, you can assign a popover to a Safari toolbar item that you have created. If the toolbar item's `command` property is set to `null`, clicking the toolbar item immediately displays the popover. If the `command` property is *not* `null`, the user must press and hold in order to see the popover—clicking alone sends the command instead. You can set a toolbar item's `command` property to `null` either programmatically or simply by leaving the toolbar item's Command field blank in Extension Builder.

You normally assign a popover to a toolbar item using Extension Builder, but you can also attach a popover to a toolbar item dynamically by setting the toolbar item's `popover` property.

Note: You can associate a toolbar item with either a menu or a popover, but not both.

Your extension can have multiple popovers. Each popover is an element in the `safari.extension.popovers` array. There is a single instance of each popover, regardless of the number of open windows. Only the active window can display the popover.

Creating the Popover Content File

A popover displays the contents of an HTML file in your extension folder. You can modify the contents of the DOM created from this file at runtime using JavaScript. If the file is specified in Extension Builder, the file loads once when the app launches. If the popover is created at runtime, the specified content file loads then.

JavaScript in the HTML file containing your popover content can make direct calls to functions in your global HTML page, using the following syntax:

```
safari.extension.globalPage.contentWindow.functionName()
```

You can also make function calls from a popover into an extension bar, but the situation is slightly more complex. There can be multiple extension bars belonging to an extension, and there is one instance of each extension bar and all its functions per open window. Your popover needs to identify which instance of which bar it wants to access, or iterate through them to access them all.

For example, if an extension has a single bar, the following snippet iterates through the extension bar instances in each open window and calls the `doSomething()` function defined in each one, but also calls the `doSomethingSpecial()` function only in the instance in the active window:

```
const myBars = safari.extension.bars;
function updateAllBars {
    for (var i = 0; i < myBars.length; ++i) {
        var barWindow = myBars[i].contentWindow;
        barWindow.doSomething();
        var myWindow = safari.application.activeBrowserWindow;
        if (myBars[i].browserWindow === myWindow) {
            barWindow.doSomethingSpecial();
        }
    }
}
```

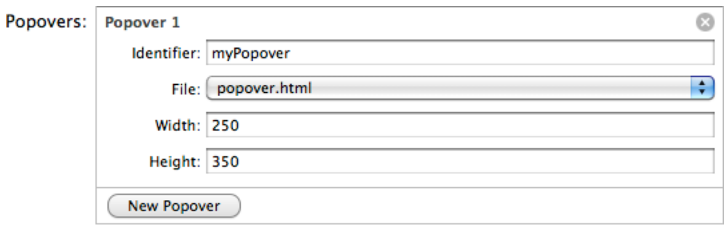
If you need to communicate with injected scripts from your popover, you must send a message to `SafariWebPageProxy`. See "Messages and Proxies" (page 87) for details.

Your popover file can reference media and data from within the extension using relative URLs or from the Internet using absolute URLs.

Setting Up Popovers in Extension Builder

To add a popover to a toolbar item, start by clicking the New Popover button in Extension Builder. The Popovers pane expands, as shown in Figure 8-1.

Figure 8-1 Popovers pane

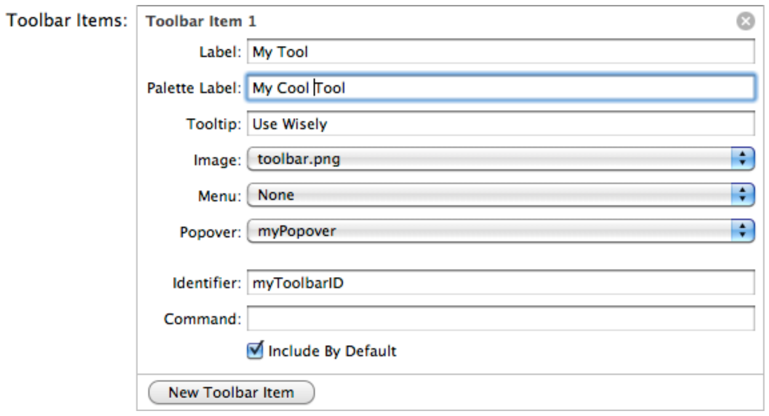

Popovers: Popover 1
Identifier: myPopover
File: popover.html
Width: 250
Height: 350

Give the popover a unique identifier and choose the HTML file for your popover content from the pop-up menu. The file must already be in your extension folder to appear in the menu.

If your popover has a fixed height and width, enter them here. If the height or width is unspecified, Safari uses a default width of 400 pixels and a default height of 300 pixels. You can change the `height` and `width` properties of your popover at runtime.

After you have added your popover in Extension Builder, go to the section for the toolbar item you want to associate with this popover and click the Popover button, then choose this popover's identifier from the pop-up menu, as illustrated in Figure 8-2.

Figure 8-2 Associating a popover with a toolbar item


Toolbar Items: Toolbar Item 1
Label: My Tool
Palette Label: My Cool Tool
Tooltip: Use Wisely
Image: toolbar.png
Menu: None
Popover: myPopover
Identifier: myToolbarID
Command:
✓ Include By Default

Responding to Events

If your popover generates its content dynamically, you should listen for `"popover"` events, and possibly `"validate"` events. In your popover HTML file or your global HTML file, register on the Safari app for the events. For example:

```
safari.application.addEventListener("validate", validateHandler, true);
safari.application.addEventListener("popover", popoverHandler, true);
```

Safari generates `"validate"` events on the toolbar item periodically, including just before your popover is displayed. Use this event if you need to dynamically enable and disable the toolbar item or update a badge on the toolbar item. In some instances, a `"validate"` event is also a good time to update your popover content. If your popover responds to `"validate"` events, check to see whether the `event.target` property for the `"validate"` event is the identifier for your toolbar item. For example:

```
if (event.target.identifier !== "myToolbarItemID") return;
```

Safari generates a `"popover"` event when your popover is about to be displayed. This is your opportunity to modify the popover content before it is seen. If your popover content does not change dynamically, you don't need to respond to this event. The `event.target.identifier` property for `"popover"` events that pertain to you is the identifier of your popover.

Handle user interaction with your popover using HTML and JavaScript, as you would in any webpage.

Creating a Popover at Runtime

Create a popover at runtime by calling `createPopover()` on the `extension` object, passing in an identifier, the URL of the content page, and optionally a width and height. For example:

```
myPop = safari.extension.createPopover(

"myPopoverID", safari.extension.baseURI + "myFile.html", width, height);
```

The `width` and `height` parameters are optional; if omitted, the default values are used.

To assign the popover to a toolbar item, set the toolbar item's `popover` property. For example:

```
myToolbarItem.popover=myPop;
```

You can obtain an instance of your toolbar item by iterating through the `extension` object's `toolbarItems` array and testing the `identifier` property for your toolbar item.

To deallocate a popover, call `safari.extension.removePopover(ID)`. Be sure to set the toolbar item's `popover` property to `null` before removing its popover.

Important: Do not attempt to deallocate a popover while it is being displayed (check the popover's `visible` property). Such an attempt fails and an exception is thrown. If necessary, you can hide the popover by calling `myPopover.hide()`.

Adding Contextual Menu Items

Contextual menus pop up when the user Control-clicks or right-clicks over an object. Safari presents different contextual menus when the mouse pointer is over the toolbar, Bookmarks bar, an extension bar, the tab bar, or the contents of a webpage.

Your extension can add menu items to the contextual menu that pops up over web content. You control the actions of the menu item by installing a listener function for the `"command"` event in either your global HTML page or in an extension bar.

Your extension can add multiple items to the contextual menu. The simplest way to add menu items is through Extension Builder, but you can also add them programmatically.

> **Note:** Adding menu items to existing Safari contextual menus is similar to, but distinct from, creating your own extension menus. See "Adding Extension Menus" (page 59) for a description of extension menus.

Context Menu Events

When the user Control-clicks or right-clicks in the web content window, a series of events are fired before the contextual menu is displayed:

1. A `"contextmenu"` DOM event that you can listen for in an injected script.

   This event gives you the opportunity to add context information to the event or to prevent the menu from displaying.
2. A `"contextmenu"` extension event that you can listen for in your global page or an extension bar.

   This event gives you the opportunity to add menu items programmatically. You can read the context information set by your injected script to help you determine what menu items to add.
3. A `"validate"` event for each menu item.

   This event gives you the opportunity to disable menu items that should not be displayed, or modify a menu item's title.

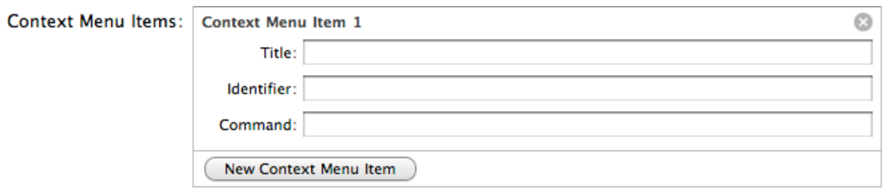
Note: There are two versions of the `"contextmenu"` event—a DOM event that you can listen for in an injected script, and an extension event that you can listen for in your global page or an extension bar. The DOM event is always sent first.

You are not required to respond to any of these events. You can add contextual menu items using Extension Builder, and if the menu items should be included in the contextual menu, you need to respond only to the `"command"` event generated when the user actually chooses one of your items from the menu. The `"contextmenu"` and `"validate"` events provide opportunities for you to modify this default behavior.

Adding a Contextual Menu Item Using Extension Builder

You add an item to the contextual menu by clicking New Context Menu Item in Extension Builder. This expands the contextual menu items pane, as shown in Figure 9-1.

Figure 9-1 Contextual Menu Items pane



Enter a title for the menu item. This is the text that will appear in the contextual menu. This field is required.

Enter an identifier. This is required. The identifier must be unique within your extension.

Enter a command name. This is the name of the `"command"` event that is generated when the user chooses your item from the menu. It does not need to be unique. For example, you might have both a toolbar item and a contextual menu item that issue the same command. If this field is left blank, the identifier is used.

Responding to Commands

When the user chooses your contextual menu item, Safari emits a `"command"` event. The `command` property of the event is the string you entered in the Command field in Extension Builder. If you left the Command field blank, the identifier is used instead.

Respond to the `"command"` event by installing a listener function in either the global HTML page or an extension bar.

You can't receive the `"command"` event in an injected script. If you need the command to initiate an action in an injected script, respond to the command in the global HTML page or an extension bar and send a message to the script. For details, see "Messages and Proxies" (page 87).

Before the contextual menu displays, Safari will ask you to validate the command by sending a `"validate"` event.

Your `"validate"` event handler should verify that it is appropriate to display the command. For example, if your menu item reloads the active tab, you should verify that the active tab has a URL to reload. If the tab is empty, your validate function should disable the menu item by setting `event.target.disabled = true`.

If you disable the contextual menu item, it is not displayed. This is different from the behavior of an extension menu item, which is displayed as grayed-out.

If there is no possibility that the command is invalid, such as a "new tab" menu item, you are not required to implement a validate handler function.

You can have multiple UI items that issue the same command, such as a toolbar button and a contextual menu item. You can use the same event handlers for command and validation, regardless of the source.

If your functions are part of the global HTML page, you should register your listener functions with the app:

```
safari.application.addEventListener("command", myCommandHandler, false);

safari.application.addEventListener("validate", myValidateHandler, false);
```

If your functions are part of an extension bar, you should register your listener functions with the extension bar's parent window:

```
safari.self.browserWindow.addEventListener("command", myCommandHandler, false);

safari.self.browserWindow.addEventListener("validate", myValidateHandler, false);
```

While you can implement the event handlers in either a global HTML page or in an extension bar, it is more efficient to use a global HTML page, because the code is loaded only once per session instead of once per page.

Modifying the Default Behavior

If you add contextual menu items using Extension Builder, the default behavior is for the items to be displayed when the user opens a contextual menu over web content, and for a `"command"` event to be generated when the user chooses a menu item. You can modify this behavior in the following ways.

Adding Context Information

If you add a listener for the DOM "`contextmenu`" event in an injected script, you can set context information by calling `setContextMenuEventUserInfo` in your event handler.

You can add the listener event to the document or any of its children, such as the body or a particular node. For example:

```
document.addEventListener("contextmenu", handleContextMenu, false);
```

The following listener function stores the element name that the user clicks, so your extension can respond differently to a click on an image or a paragraph, for example:

```
function handleContextMenu(event) {
    safari.self.tab.setContextMenuEventUserInfo(event, event.target.nodeName);
}
```

The data you store can be retrieved from the `userInfo` property of later events. You can store any data as user info that you can pass in a message.

Disabling the Contextual Menu

You can prevent the contextual menu from displaying at all by calling `event.preventDefault()` from an injected script in response to the DOM "`contextmenu`" event.

The following snippet prevents the contextual menu from displaying if the user clicks a video element:

```
document.addEventListener("contextmenu", handleContextMenu, false);

function handleContextMenu(event) {
    if (event.target.nodeName == "VIDEO") {
        event.preventDefault();
    }
}
```

Adding Contextual Menu Items Programmatically

You can add menu items to the contextual menu by responding to the extension version of the `"contextmenu"` event in your global page or an extension bar. If you stored information on the event by calling `setContextEventUserInfo()` in your injected script, you can use that information to help you decide what menu items to add.

For example, the following snippet adds an Enlarge Image menu item to the contextual menu if the user clicks an image. (This snippet relies on an injected script to store the event target's node name as user info.)

```
safari.application.addEventListener("contextmenu", handleContextMenu, false);


function handleContextMenu(event) {
    if (event.userInfo === "IMG") {
        event.contextMenu.appendContextMenuItem("enlarge", "Enlarge Item");
    }
}
```

Note: You can only add menu items in response to the `"contextmenu"` event, not delete them. To temporarily delete a menu item, set `event.target.disabled = true` in response to the menu item's `"validate"` event.

Changing the Contextual Menu Item Title

You can modify a menu item's title before it is displayed by setting the `event.target.title` property in a `"validate"` event handler.

If you set user info in an injected script using a `"contextmenu"` event handler, you can read the `event.userInfo` property in your `"validate"` event handler to help decide how to change the label.

For example, if an injected script stores the selected text being clicked, you could change a Search Google Scholar menu item to include the selected text using the following snippet:

```
event.target.title = "Search for \u201C" + event.userInfo + "\u201D on Google
Scholar";
```

In this example, you would also want to use the `event.userInfo` in your `"command"` event handler.

Deciding Where to Respond

When your contextual menu item is chosen, a `"command"` event is generated. You can listen for the event in either a global HTML page or in an extension bar. If you put the event handler in the global file, you should register for the event at the app level. If your event handler is in an extension bar, you should register with the extension bar's parent window.

The difference is that there is only one instance of the global HTML page's functions, but there is an instance of an extension bar in every open window.

If every instance of the extension bar registers for events at the app level, every instance responds to the command. If each instance registers with its parent window, only the instance in the window where the item was chosen responds to the command.

Using the global file is more efficient, as it loads only once. Furthermore, you shouldn't create an empty extension bar just to hold an event handler. If your extension has a bar, however, it might make sense to put the event handler there, particularly if the action it takes is window-specific, like rearranging the tabs.

The only time you might want to put an event handler in the extension bar and register it with the app is if your command acts on all open windows. Then your choice would be to iterate through the windows in a global function or have a function local to each window that responds to the event independently.

If You Respond from a Global HTML Page

- Register with the app: `safari.application.addEventListener()`.
- The window the event came from is `safari.application.activeBrowserWindow`.

If You Respond from an Extension Bar

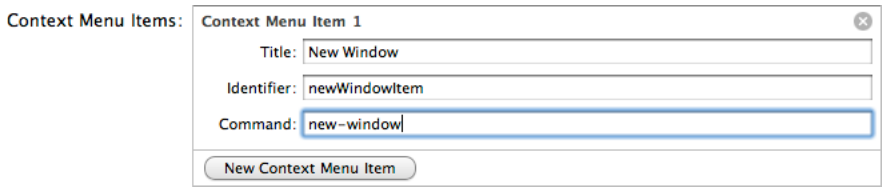
- Register with the parent page: `safari.self.browserWindow.addEventListener()`.
- The window the event came from is `safari.application.activeBrowserWindow`.

Example: Implementing a New Window Contextual Menu Item

The following example adds a New Window item to the contextual menu, which opens a new browser window.

You add the menu item in Extension Builder, as illustrated in Figure 9-2.

Figure 9-2 Adding a menu item


Context Menu Items:
Context Menu Item 1
Title: New Window
Identifier: newWindowItem
Command: new-window

The code in the following listing responds to the `new-window` command event and the `new-window` validate event. Since opening a new window is possible any time, there is no validate handler. The `"command"` listener function is registered with the app.

Listing 9-1 (page 75) is intended to be included in your global HTML page. If it is included in an extension bar, the event listeners need to be added to the bar's parent window instead of the app. Otherwise each instance of the extension bar executes the command—instead of adding just one window, every open window adds a window, and the number of windows is doubled.

Listing 9-1 New window command handlers

```
function performCommand(event) {
    if (event.command === "new-window") {
        safari.application.openBrowserWindow();
    }
}
safari.application.addEventListener("command", performCommand, false);
```

Injecting Scripts

You can inject `.js` files into webpages (a `.js` file is a text file with the `.js` extension, containing JavaScript functions and commands). The scripts in these files have access to the DOM of the webpages they are injected into. Injected scripts have the same access privileges as scripts executed from the webpage's host.

About Injected Scripts

Injected scripts are loaded each time an extension-accessible webpage is loaded, so you should keep them lightweight. If your script requires large blocks of code or data, you should move them to the global HTML page. For details, see "Example: Calling a Function from an Injected Script" (page 90).

An injected script is injected into every webpage whose URL meets the access limitations for your extension. For details, see "Access and Permissions" (page 95).

Scripts are injected into the top-level page and any children with HTML sources, such as iframes. Do not assume that there is only one instance of your script per browser tab. If you want your injected script not to execute inside of iframes, preface your high-level functions with a test, such as this:

```
if (window.top === window) {
   // The parent frame is the top-level frame, not an iframe.
   // All non-iframe code goes before the closing brace.
```

Injected scripts have an implied namespace—you don't have to worry about your variable or function names conflicting with those of the website author, nor can a website author call functions in your extension. In other words, injected scripts and scripts included in the webpage run in isolated worlds, with no access to each other's functions or data.

Injected scripts do not have access to the `safari.application` object. Nor can you call functions defined in an extension bar or global HTML page directly from an injected script. If your script needs to access the Safari app or operate on the extension—to insert a tab or add a contextual menu item, for example—you can send a message to the global HTML page or an extension bar. For details, see "Messages and Proxies" (page 87).

Important: When you use `safari.extension` from within an injected script, you are not addressing the `SafariExtension` class. You are addressing the `SafariContentExtension` class.

Your injected scripts can access resources—images, HTML, and other scripts, for example—within your extension folder. Relative URLs are relative to the webpage your script is injected into, however. If you need to access local resources, use `safari.extension.baseURI +`"relative path and filename". You cannot access resources on the user's hard drive outside of the extensions folder.

To add content to a webpage, use DOM insertion functions, as illustrated in Listing 10-1.

Listing 10-1 Modifying a webpage using DOM insertion

```
// create a para and insert it at the top of the body
var newElement = document.createElement("p");
newElement.textContent = "New Element!";
newElement.style.color = "red";
newElement.style.float = "right";
document.body.insertBefore(newElement, document.body.firstChild);
```

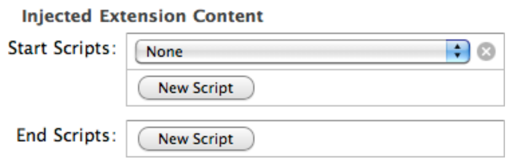
Note: DOM modification should take place in an End Script; in a Start Script, `document.body` may be undefined.

Adding a Script

To add an injected script, follow these steps:

1. Create an extension folder—open Extension Builder, click +, choose New Extension, give it a name and location.
2. Drag your script file into the extension folder.
3. Click New Script under Injected Extension Content in Extension Builder, as illustrated in Figure 10-1.

Figure 10-1 Specifying injected content



You can choose to inject your script as a Start Script or an End Script. An End Script executes when the DOM is fully loaded—at the time the `onload` attribute of a `body` element would normally fire. Most scripts should be injected as End Scripts.

A Start Script executes when the document has been created but before the webpage has been parsed. If your script blocks unwanted content it should be a Start Script, so it executes before the page displays.

4. Choose your script file from the pop-up menu.

You can have both Start and End Scripts. You can have more than one script of each type.

In order for your scripts to be injected, you must specify either Some or All website access for your extension. You can have your script apply to a single webpage, all webpages, or only certain webpages—pages from certain domains, for example. For details, see the description of whitelists and blacklists in "Access and Permissions" (page 95).

Injecting Styles

You can designate CSS style sheets to be injected into websites. Your style sheets can add to or override styles provided by the website author.

About Injected Style Sheets

Injected style sheets are treated as user style sheets, as defined by the W3C. This means that first your injected styles are defined, then the author's styles are added, then any of the author's properties declared as `!important` are added, then your properties defined as `!important` are added. At each stage, a new definition overrides any previous one.

This means that style properties in your injected style sheets are added to existing page style properties, but do not override them, unless you declare them as `!important`.

For example, to override a website using colored text on a colored background and set it to black text on a white background, you could add these styles:

```
body {
     color:black !important;
     background:white !important;
     }
```

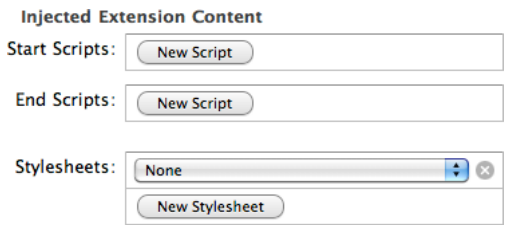
Adding a Style Sheet

To add an injected style sheet, follow these steps:

1. Create an extension folder—open Extension Builder, click +, choose New Extension, give it a name and location.
2. Drag your style sheet into the extension folder.

3. Click New Style Sheet under Injected Extension Content in Extension Builder, as illustrated in Figure 11-1.

Figure 11-1 Specifying an injected style sheet



4. Choose your style sheet from the pop-up menu.

You can have more than one injected style sheet.

In order for your style sheets to be injected, you must specify either Some or All website access for your extension. You can have your style sheet apply to a single webpage, all webpages, or only webpages from certain domains. For details, see the description of whitelists and blacklists in "Access and Permissions" (page 95).

If you reference images or other external resources in your style sheets, you can use relative URLs to indicate sources within your extension package, relative to the style sheet.

Important: In injected scripts, relative URLs are relative to the webpage the script is injected into. In injected style sheets, relative URLs are relative to the style sheet.

The Windows and Tabs API

The standard `window.open()` JavaScript method cannot be used to open a new tab and window from a global HTML file or an extension bar. Instead, the global file and extension bars have access to the `SafariApplication`, `SafariBrowserWindow`, and `SafariBrowserTab` classes, whose methods and properties allow you to work with windows and tabs.

In Safari 5.1 and later, there are additional events generated:

- When a window or tab is opened
- Before a window or tab is closed
- When a window or tab is activated
- Before a window or tab is deactivated
- Before a search is performed from the Smart Search Field *(only in Safari 6.0 and later)*
- Before navigating to a new URL
- After navigating to a new URL (when the main frame is loaded)
- Before Safari's Private Browsing mode is activated or deactivated *(only in Safari 6.0 and later)*

Navigation occurs whenever the user loads a new page in any manner—clicking a link in a tab or window, opening a bookmark, entering a URL in the address field, or typing a query in the web search field.

SafariApplication

There is one instance of the `SafariApplication` class: `safari.application`. The `SafariApplication` instance has a method for opening browser windows:

```
var myWin = safari.application.openBrowserWindow();
```

All open browser windows can be accessed as properties of the `SafariApplication` instance: `safari.application.browserWindows` is an array of all open windows, and `safari.application.activeBrowserWindow` is the currently active browser window.

SafariBrowserWindow

Each open browser window is an instance of the `SafariBrowserWindow` class, which has methods to activate windows, close them, and determine if a window is visible onscreen. The `SafariBrowserWindow` class also gives direct access to tabs.

- `browserWindow.tabs`—returns an array of all the tabs in the window, in left-to-right order.
- `browserWindow.openTab()`—creates a new tab at any point in the array. The tab can be hidden.
- `browserWindow.insertTab()`—inserts an existing tab at any point in the array.
- `browserWindow.activeTab`—returns the currently selected tab in the window.

A browser window is commonly accessed as a property of the Safari app instance:

```
safari.application.browserWindows[n]
```

or:

```
safari.application.activeBrowserWindow
```

SafariBrowserTab

The `SafariBrowserTab` class allows you to identify a tab's parent browser window, get and set the URL associated with a tab, make a tab active, close a tab, and extract a `data://` URL containing a snapshot image of what is rendered in the tab as a base-64 encoded PNG.

For example, this opens a window and returns the active tab:

```
var newTab = safari.application.openBrowserWindow().activeTab;
```

And this opens a new tab in the window containing the extension bar:

```
var newTab = safari.self.browserWindow.openTab();
```

Events

In Safari 5.1 and later, you can listen for and respond to the following window and tab events:

- Open—Safari sends an `"open"` event to a window or tab when it is first opened.
- Close—Safari sends a `"close"` event to a window or tab when it is about to close.
- Activate—Safari sends an `"activate"` event to a window or tab whenever it is activated.

- Deactivate—Safari sends a "`deactivate`" event to a window or tab when it is about to be deactivated.
- Before Navigate—Safari sends a "`beforeNavigate`" event to a tab whenever a new URL is about to load there. You can prevent Safari from loading the URL by calling `preventDefault()` on the event. The event has a `url` property that you can inspect to see where the navigation event leads.
- Navigate—Safari sends a "`navigate`" event to a tab when the main frame of the new URL has loaded.
- Before Search—Safari sends a "`beforeSearch`" event to a tab whenever Safari is about to start a search and show the results in that tab. You can intercept the query and execute an alternate task *(only in Safari 6.0 and later)*

You can listen for these events on the `application` object from an extension bar or your global HTML page. You can also listen on a specific window or tab in the app.

Important: These events do not bubble, so either listen for them on the target tab or window, or else set the capture parameter to `true` when installing your event listener. For example:

```
safari.application.addEventListener("open", openHandler, true);
myTab.addEventListener("close", closeHandler, false);
```

Working with Safari Reader

Reader is a Safari feature that allows users to read online articles in a continuous, clutter-free view, with no ads or visual distractions. Reader concatenates multipage articles into a single scrolling pane. In Safari 5.1 and later, there are events that fire when Reader is available, activated, or deactivated. In your extension, you can enter and exit Reader programmatically, inspect properties on the Reader object, and inject scripts and style sheets.

Reader Events

Safari generates the following Reader events on the relevant tab:

- Available—an `"available"` event on a tab when Reader is available to be invoked.
- Activate—an `"activate"` event on a tab when Reader is invoked.
- Deactivate—a `"deactivate"` event on a tab when Reader is dismissed.

You can listen for these events on the tab itself, or on the window or app. Only the `"available"` event bubbles, however, so either listen on the tab or set the `capture` parameter to `true` when installing your listener functions. For example:

```
safari.application.addEventListener("activate", activeHandler, true);

tab.addEventListener("available", availHandler, true);

tab.addEventListener("deactivate", deactivateHandler, false);
```

In your `"available"` event handler, you should test the event target to make sure that it is a Reader instance that has become available, and not something else. See Listing 13-1 (page 84):

Listing 13-1 Reader “available” handler

```
function availHandle(event) {
    if (event.target instanceof SafariReader) {
        myReader = event.target;
        myReader.enter();
    }
}
```

Note: There is no `"unavailable"` event for Reader. Once Reader is available, it remains available until the next `"navigate"` event.

Reader Methods and Properties

The Reader object is an instance of the `SafariReader` class. You can invoke the `enter()` and `exit()` methods to create or dismiss a Reader view.

If a `SafariReader` instance is available, you can obtain it by getting the `reader` property of the tab that received the `"available"` event. For example:

```
theReader = myTab.reader;
```

Because the event target of a Reader `"available"` event is a tab, you can also obtain a Reader instance from the event target, as shown in Listing 13-1 (page 84).

Once you have an instance of Reader, you can open a Reader view by calling `theReader.enter();`.

To close the Reader view, call `theReader.exit();`.

You can also inspect properties of a Reader instance, specifically `available`, `visible`, and `tab`. The `available` property generates an event when it becomes true, and the `visible` property generates `"activate"` and `"deactivate"` events when it changes, so there is normally no need to poll these properties.

The `tab` property contains the tab the Reader instance is available or active within.

For more information, see *SafariReader Class Reference* .

Injecting Style Sheets and Scripts into Reader

In Safari 6.0 and later, you can automatically inject style sheets and scripts into Reader by specifying the safari-reader scheme in the whitelist and blacklist in Extension Builder. See "Whitelists and Blacklists" (page 97) for more information.

If you need to dynamically add a new style sheet or script, you can use the `addContentStyleSheet` and `addContentScript` APIs.

Injecting into Reader Dynamically

To inject into Reader dynamically, you can call `addContentStyleSheetFromURL` and `addContentScriptFromURL` on the `SafariExtension` object using the whitelist and blacklist to limit the scope of the style sheet or script to only Reader pages.

```
safari.extension.addContentStyleSheetFromURL("pathToYourStylesheet",
["safari-reader://*/*"]);
```

See “Whitelists and Blacklists” (page 97) to see how to work with whitelists and blacklists.

Similarly, to remove a style sheet or script from a Reader page, you can call `removeContentStyleSheet` or `removeContentScript` on the `SafariExtension` object:

```
safari.extension.removeContentStyleSheet("pathToYourStylesheet");
```

Adding or removing style sheets applies to all pages immediately. Adding or removing a script, however, only applies to pages opened or reloaded after that point.

Sending Messages to Scripts Injected into Reader

Like your style sheets, your scripts are injected into every webpage to which your extension has access. When dealing with injected scripts in Reader pages, the only difference is that when you want to message an injected script from, for example, the global HTML page, you must use the `dispatchMessage` method on the `Reader` object.

```
// Example of dispatching a message to a Content Script injected into Reader.
    var browserWindows = safari.application.browserWindows;
    for (var i = 0; i < browserWindows.length; ++i) {
        var tabs = browserWindows[i].tabs;
        for (var j = 0; j < tabs.length; ++j) {
            var tab = tabs[j];
            if (!tab.reader.visible)
                continue;
            tab.reader.dispatchMessage("Message to the Content Scripts injected
into Reader");
```

Messages and Proxies

Code in your global HTML page and extension bars interacts with the Safari app; it can't directly access the contents of a webpage loaded in a browser tab. Similarly, an injected script interacts with web content; it can't access the same Safari extensions API as extension bars or a global page.

But it's sometimes desirable to cross this boundary. You may have controls in an extension bar or the main Safari toolbar that you want to affect web content, for example, or you may have a large block of code or data in an extension bar or your global HTML page that you want to use from an injected script.

The solution is to pass messages between parts of your extension. Because your global HTML page and extension bars can't address webpages directly, they send messages to the `SafariWebPageProxy`. Similarly, injected scripts can't address the global HTML page or an extension bar directly, so they send messages to the `SafariContentBrowserTabProxy`.

Message Structure

A message is an event whose type is `"message"`. You send a message by calling `dispatchMessage(name, data)` and receive messages by registering a listener function for `"message"` events.

A message event has a `name` property and a `message` property, which are the name and data you pass in `dispatchMessage`.

This can be a little confusing, so it bears repeating: `event.name` is the message name, and `event.message` is the message data.

Message data is not limited to a single data type; it can be Boolean, numeric, a string, an array, a RegExp object, or anything that conforms to the W3C standard for safe passing of structured cloned data. It can also be null, undefined, or left blank, in cases where the command needs no data.

For example, the following snippet sends an array in a message:

```
var myArray = ["a", "b", "c"];
safari.self.tab.dispatchMessage("passArray", myArray);
```

Sending Messages to an Injected Script

To send messages to an injected script, you call the `SafariWebPageProxy` object's `dispatchMessage()` method. The proxy stands in for the web content, which can be accessed as the `page` property of a `SafariBrowserTab` object, which is in the `tabs` array or `activeTab` property of a `SafariBrowserWindow` object. So, sending a message to a script takes the general form:

```
safari.application.activeBrowserWindow.activeTab.page.dispatchMessage("name",
"data");
```

Another example would be:

```
safari.self.tabs[0].page.dispatchMessage(myMessageName,myData);
```

Important: When dispatching a message to a page in Safari 6.1 and later, the message could be sent to pages not currently visible. The Safari application instance could also receive messages from multiple pages that target the same tab. For more information, read "Tabs in Safari 6.1 and Later" (page 89).

The second example sends a message to the page in the leftmost tab of the window containing the extension bar.

In order to receive the message, your injected script must have a listener function defined and registered for `"message"` events. The listener function is called for all messages, so it needs to check the message name to be sure it's responding to the desired message. For example, the following function looks for an `activateMyScript` message, then parses the message content:

```
function handleMessage(msgEvent) {
    var messageName = msgEvent.name;
    var messageData = msgEvent.message;
    if (messageName === "activateMyScript") {
        if (messageData === "stop") {
            stopIt();
        }
        if (messageData === "start") {
            startIt();
        }
    }
}
```

The listener function in an injected script is added as a listener for `"message"` events in the `SafariContentWebPage` object (`safari.self`):

```
safari.self.addEventListener("message", handleMessage, false);
```

Note: The `addEventListener()` method takes three parameters: a string for the event type, the name of the listener function (not in quotes, and without the usual trailing `()`), and a Boolean that tells the system whether the listener function should be given the event in the capture phase or the bubble phase. The Boolean value is usually set to `false`.

Tabs in Safari 6.1 and Later

Safari 6.1 and later may preload webpages in the background to improve the user experience. Therefore, it is possible for each tab to be associated with more than one webpage, some of which are hidden from the user.

When dispatching messages to the `page` property of `SafariBrowserTab` objects in Safari 6.1 and later, Safari dispatches your message to all pages associated with that tab, including pages not visible to the user. Updating the UI of a non-visible page impacts battery life and does not benefit the user. If your extension updates the UI of the page, first ensure that the page is visible to avoid executing code on hidden webpages.

You ensure the current page is visible by using the Page Visibility API. Query that `document.hidden` is false before running code on the webpage. Also listen for the `visibilitychange` JavaScript event to determine whether the visibility of the page has changed.

Receiving Messages from an Injected Script

A receiver function in an extension bar or global HTML page behaves identically to a receiver function in a script (it checks the message name before evaluating the message). But instead of registering with the `SafariContentWebPage`, a receiver function in an extension bar or global HTML page can register for the event at the tab, window, or app level:

```
safari.application.activeBrowserWindow.activeTab.addEventListener("message",
waitForMessage, false);
safari.application.activeBrowserWindow.addEventListener("message", waitForMessage,
 false);
safari.application.addEventListener("message", waitForMessage, false);
```

The message is sent first to the app, then filters down to the window and tab. At each level, the event is sent to listener functions registered with Boolean `true`. If no one has claimed it, the message then bubbles up from the tab through the window and back to the app, this time for listeners registered with Boolean `false`. In most cases, it makes no practical difference at which level you intercept the message.

To send a message to an extension bar or global HTML page from an injected script, the script dispatches the message to the tab proxy:

```
safari.self.tab.dispatchMessage("heyExtensionBar","Klaatu barada nikto");
```

Example: Calling a Function from an Injected Script

If an injected script makes use of a large block of code or an extensive table of data, it is more efficient to put the bulky code or data in an extension bar or a global HTML page than in the injected script.

The following example is an injected script, Listing 14-1 (page 90), that makes a function call to a global HTML page, Listing 14-2 (page 91), using messages. To see the example in action, follow these steps:

1. Create an extension folder using Extension Builder.
2. Copy the listings into a text editor and save as `Injected.js` and `Global.html`.
3. Drag `Injected.js` and `Global.html` into your extension folder.
4. Click Extension Global Page in Extension Builder and choose `Global.html`.
5. Click New Script in End Scripts and choose `Injected.js`.
6. Set the Extension Website Access level to All.
7. Click Install.

Listing 14-1 Injected.js

```
var initialVal=1;
var calculatedVal=0 ;

function doBigCalc(theData) {
    safari.self.tab.dispatchMessage("calcThis",theData);
}

function getAnswer(theMessageEvent) {
    if (theMessageEvent.name === "theAnswer") {
        calculatedVal=theMessageEvent.message;
        console.log(calculatedVal);
    }
}
safari.self.addEventListener("message", getAnswer, false);

doBigCalc(initialVal);
```

Listing 14-2 Global.html

```
<!DOCTYPE HTML>
<html>
<head>
<title>global HTML page</title>
<script type="text/javascript">

function bigCalc(startVal, event) {
    // imagine hundreds of lines of code here...
    var endVal = startVal + 2;
    // return to sender
    event.target.page.dispatchMessage("theAnswer", endVal);
}

function respondToMessage(theMessageEvent) {
    if(theMessageEvent.name === "calcThis") {
        var startVal=theMessageEvent.message;
        bigCalc(startVal, theMessageEvent);
    }
}

    safari.application.addEventListener("message",respondToMessage,false);
</script>
</head>
<body>
</body>
</html>
```

In Listing 14-1 (page 90), the final value of the calculation is logged to the webpage console. To see the log entry, choose Show Web Inspector in the Develop menu.

For an example that shows how to pass messages to a script from an extension bar, see "Message-Passing Example" (page 42).

Blocking Unwanted Content

Safari 5.0 and later (and other WebKit-based browsers) generates a `"beforeload"` event before loading each subresource belonging to a webpage. The `"beforeload"` event is generated before loading every script, iframe, image, or style sheet specified in the webpage, for example.

To block content, your script must be run as a Start Script, so that it executes before the content is displayed.

If your script responds to a `"beforeload"` event by calling `event.preventDefault()`, the pending subresource is not loaded. This is a useful technique for blocking ads, as shown in Listing 14-3.

Listing 14-3 Blocking content

```
function blockAds() {
    var itsAnAd = event.url.match(/ads.example.com/i);
    if (itsAnAd) {
        event.preventDefault();
    }
}

document.addEventListener("beforeload", blockAds, true);
```

Note: The `url` property of the event is the URL of the pending resource.

Blocking unwanted content can require a fair amount of code or a large table of data (or both) to filter all unwanted content accurately. You should put large blocks of code or data in your global page, so they are loaded only once, not in a script that loads before every webpage.

This creates a problem. You can put the code in your global page and call it by passing a message, but `dispatchMessage()` is an asynchronous function call. You need to delay the resource load until you know whether to allow it.

To solve this problem, use the `canLoad()` function, which returns data and operates synchronously:

```
var myReply = safari.self.tab.canLoad(event, myMessageData);
```

This dispatches a message event synchronously. You pass the `"beforeload"` event and any message data. The name of the message is always `"canLoad"`. The message data can be anything you like, but probably includes the URL of the resource in question.

A listener function in your global page (or an extension bar) sees the `"canLoad"` message, determines whether the resource should be blocked, and sets `event.message` to a reply that tells your injected script what to do. The reply can be a string, an object, or anything that can be passed as message data.

You need to register a listener function for the `"beforeload"` event in your injected script and call `canLoad()` from the listener function, so it all takes place before the resource loads. To block a resource from loading, call `event.preventDefault()` from the listener function as well.

The following example listens for the `"beforeload"` event in an injected script and passes the resource URL to the global page in `canLoad()`. The listener function in the global page compares the URL to a list of domains to exclude, and sets the message returned to `"allow"` or `"block"`. The listener function in the injected script waits for the returned value, then conditionally prevents the load.

This part goes in your injected Start Script:

```
function isItOkay() {
    var myMessageData = event.url;
    var theAnswer = safari.self.tab.canLoad(event, myMessageData);
    if (theAnswer == "block") {
        event.preventDefault();
    }
}

document.addEventListener("beforeload", isItOkay, true);
```

This part goes in your global HTML page:

```
function blockOrAllow(event) {
    if (event.name === "canLoad") {
        var itsAnAd = event.message.match(/ads.example.com/i);
        if (itsAnAd) {
            event.message = "block";
        }
        else {
            event.message = "allow";
        }
    }
}

safari.application.addEventListener("message", blockOrAllow, true);
```

Access and Permissions

Extensions can have two parts—an app part, consisting of any global page or extension bars, and a content part, consisting of any injected scripts or style sheets. The two parts have different access and permissions.

In addition, there are settings you can specify when building your extension that select the websites your extension can interact with.

Note: For security reasons, there are some things that no part of your extension can access. This includes files on the user's hard disk outside of the extension package, as well as functions and variables defined in scripts loaded from the webpage's domain.

The Global HTML Page, Popovers, and Extension Bars

The global HTML page, popovers, and extension bars have access to the `SafariApplication` and `SafariExtension` classes. They can work with windows and tabs, extension settings, and add or remove extension items. They can also respond to commands from the Safari toolbar or the contextual menu that appears over a webpage.

The global HTML page, popovers, and extension bars do not have access to the content of webpages, and they can communicate with injected scripts only by sending messages—they cannot access an injected script's functions or variables directly.

The global page, popovers, and extension bars do not have permission to use the JavaScript `window.open()` method. They must use the Safari Extensions API. See "The Windows and Tabs API" (page 81).

Injected Scripts and Style Sheets

Injected scripts have access to the `SafariContentExtension` class. They have the same permission to access and modify the webpages they are injected into as scripts originating in the webpage's own domain. They have permission to use the standard JavaScript API, as well as Safari-specific and WebKit-specific JavaScript APIs.

Injected scripts cannot access the `SafariApplication` or `SafariExtension` classes. They cannot respond to command events generated by the Safari toolbar or contextual menus, nor can they access functions or variables defined in the global HTML page or extension bars. They can, however, send messages to the global HTML page and extension bars, and the message data can be an object (such as an array, for example) declared in the injected script.

Injected scripts and style sheets cannot access resources within the extension folder, such as images or other files, using relative URLs. Any relative URL in an injected script or style sheet is interpreted as relative to the webpage. To access resources within the extension folder from an injected script or style sheet, you must use an absolute URL. For details, see "Accessing Resources Within Your Extension Folder" (page 32).

Extension Website Access

You choose the webpages and domains your extension has access to in Extension Builder. Only the websites you choose have web content injected into them, and only those websites can be manipulated using the `tab` object's properties, such as `title` and `url`.

Use the Extension Website Access field in Extension Builder to restrict your extension's access to external websites. Your choices are as follows:

- None—Your extension cannot access webpages by injecting scripts or style sheets, and most tab properties are undefined.
- Some—Your extension can access webpages from a list of domains.

  You are prompted for a list of domain patterns. For example: `developer.apple.com` or `www.example.org.jp`.

  A leading `*` character matches any string in the domain. For example: `*.apple.com` matches `www.apple.com`, `developer.apple.com`, or any host name in the `apple.com` domain. Similarly, `*.co.jp` matches all `co.jp` domains and `*.jp` matches all `.jp` domains.

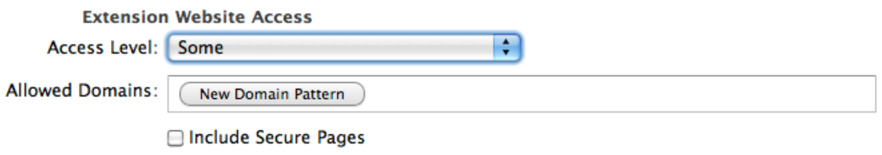
  > **Note:** Do *not* include a scheme, such as `http://` in the domain pattern.

- All—Your extension's access is not limited by a primary list of domain patterns. Potentially, your extension has access to all domains. Website access can be limited by using a whitelist and blacklist, however. See "Whitelists and Blacklists" (page 97).

Important: If you set your access to Some, and do not specify any domain patterns, your extension has no website access.

If you choose Some or All, you can further choose to allow your extension access to secure sites (HTTPS URLs) or not, as shown in Figure 15-1.

Figure 15-1 Access to secure pages



Whitelists and Blacklists

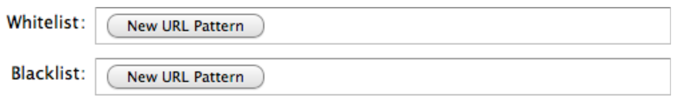
The whitelist and blacklist work in conjunction with the Extension Website Access field. First, access is limited by the Extension Website Access settings, then the whitelist and blacklist are applied.

- If there is no whitelist or blacklist, no restrictions are added to your Extension Website Access.
- If there is a whitelist, your scripts and styles are applied only to webpages whose URL match an entry on your whitelist.
- If there is a blacklist, your scripts and styles are not applied to any webpages whose URL matches a blacklist entry.

Again, note that these restrictions are in addition to those set in the Extension Website Access field. If you specify Some access, for example, you have access only to the domains matching your provided domain patterns. Items in your whitelist and blacklist create additional restrictions within those domains. Be sure all the items in your whitelist are within a domain you have access to.

Add URLs to the whitelist or blacklist by clicking New URL Pattern, as illustrated in Figure 15-2.

Figure 15-2 Whitelist and Blacklist



A URL pattern takes the form *Scheme://Domain/Path* .

Scheme can be `http` or `https`.

Domain is the host domain, such as `developer.apple.com` or `www.example.co.jp`.

Path is the directory or webpage, such as `safari/` or `safari/library/navigation/index.html`.

A URL pattern can include the * character to match any string. This allows you to specify all pages in a particular domain, for example, without having to create an exhaustive list.

The * character can be used anywhere in the domain or path, but not the scheme. Here are some examples:

- `http://*/*`—matches all http URLs.
- `http://*.apple.com/*`—matches all webpages from apple.com.
- `http://developer.apple.com/*`—matches all webpages from developer.apple.com.
- `https://secure.example.com/accounts/*`—matches all webpages from the accounts directory of secure.example.com that are delivered over HTTPS.
- `http://www.example.com/thepath/thepage.html`—matches one webpage

Important: The format for URL patterns in a whitelist or blacklist is not the same as the format for domain patterns in Extension Website Access.

Settings and Local Storage

You can define settings for your extension using Extension Builder. You can choose a type of user interface—such as a checkbox, radio button, text field, or slider—and a default value for each setting. You can choose to make any setting secure (encrypted).

The settings you define appear in your extension's preference pane, in Safari preferences. Safari handles the user interface, stores the values, and notifies you when a value changes.

There is also an API for accessing your settings programmatically. The API provides for both normal and secure (encrypted) settings. The API is similar to the HTML5 local storage API, but the settings API has an additional feature: support for default values.

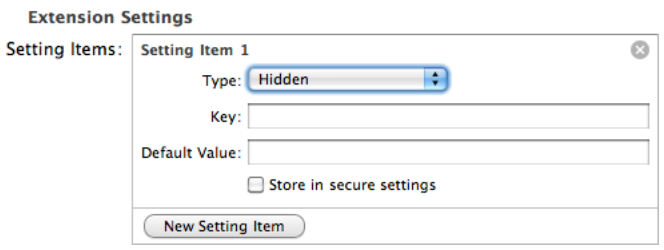
You can also make use of HTML5 client-side data storage, commonly referred to as local storage. You can use both Safari settings and HTML5 local storage if you like.

How to Create User Settings

You create your extension's user settings and define the user interfaces for them in Extension Builder. Click New Setting Item under Extension Settings to begin.

The extension settings pane expands, as shown in Figure 16-1.

Figure 16-1 Settings pane



The pane changes depending on the type of setting you choose, but you are usually prompted for a key, a default value, and a title, along with the option of saving the item in secure settings.

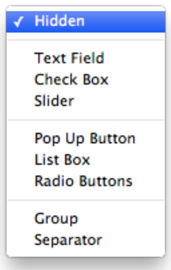
The key is the identifier for the item, used in the settings API.

The default value is the initial value for the item when your extension is installed. A default value is optional.

The title is the label the user sees for the setting.

Use the Type pop-up menu to choose the user interface control type. The menu is shown in Figure 16-2.

Figure 16-2 Setting types

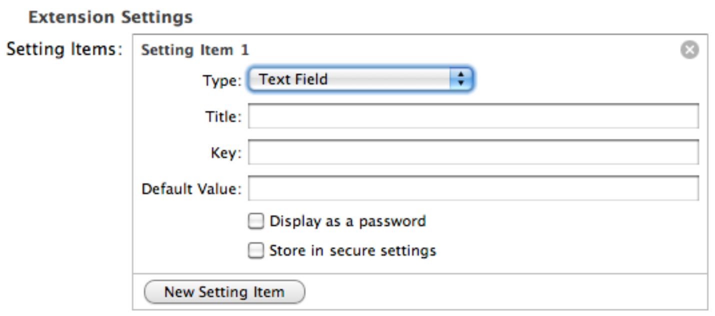


Hidden Settings

Hidden settings have no title and are not displayed in the extension's settings pane. They are for your internal settings that you intend to handle programmatically. The reason you might want to define a hidden setting in Extension Builder is to give it a default value.

Text Field Settings

Text field settings take a string as a value and have the option of being displayed as a password (characters are not visible after entry).


Extension Settings

Setting Items: Setting Item 1

Type: Text Field

Title:

Key:

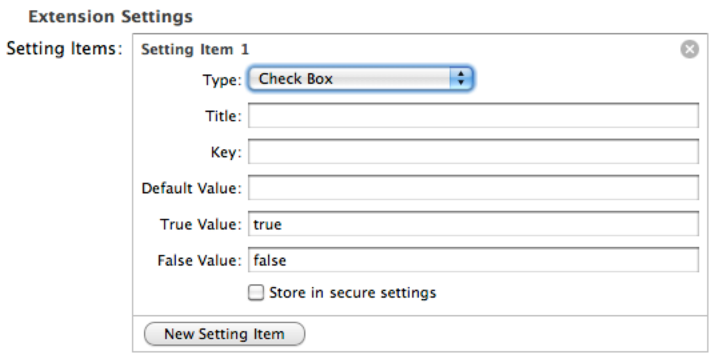
Default Value:

☐ Display as a password

☐ Store in secure settings

Checkbox Settings

A checkbox is true when checked, false when unchecked. But you can assign any pair of values to a checkbox's two states.

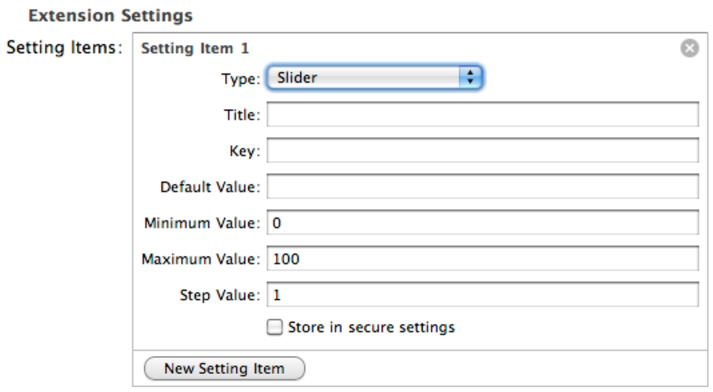


Type:	Check Box
Title:	
Key:	
Default Value:	
True Value:	true
False Value:	false

☐ Store in secure settings

Slider Settings

Sliders have a minimum value at the far left, a maximum value at the far right, and step value (the smallest change possible for the control).

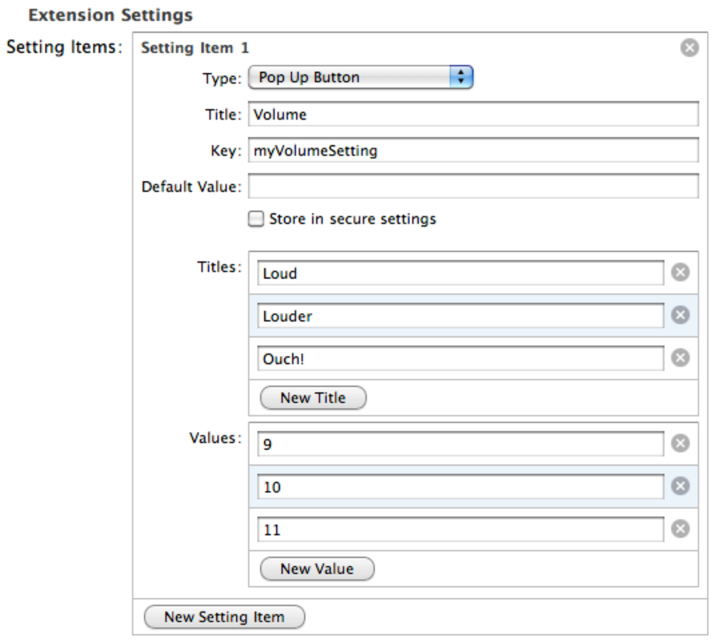


Type:	Slider
Title:	
Key:	
Default Value:	
Minimum Value:	0
Maximum Value:	100
Step Value:	1

☐ Store in secure settings

Pop-Up Button Settings

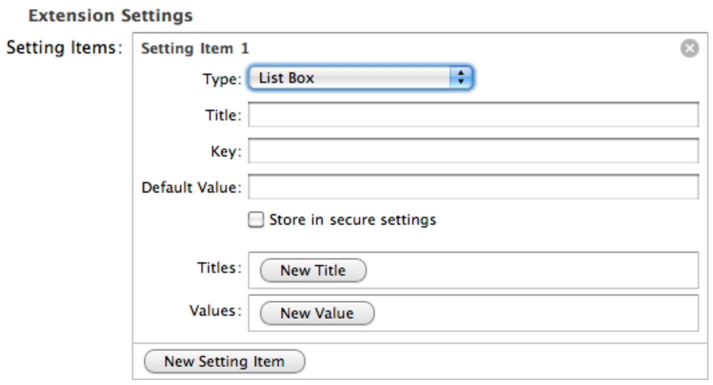
A pop-up button displays the title of the currently selected item. When the user clicks the button, a pop-up menu presents a list of all the items, allowing the user to choose one. The button itself has a title, a key, and a value. The list items have titles and values.



Type: Pop Up Button
Title: Volume
Key: myVolumeSetting
Default Value:
Store in secure settings
Titles: Loud, Louder, Ouch!
Values: 9, 10, 11

List Box Settings

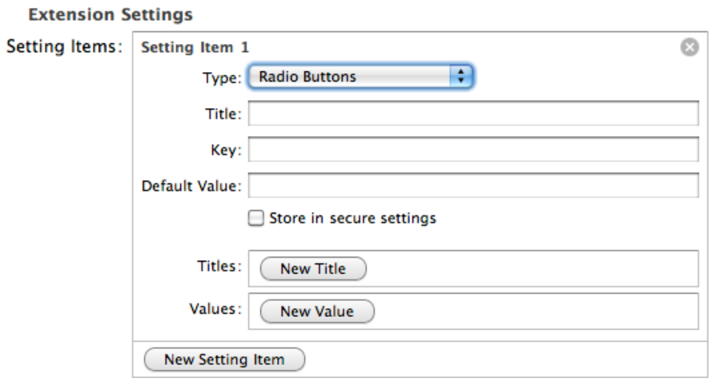
List box settings contain a list of items, such a file list. Each item has a title the user sees and a value that the list box returns when that item is chosen. The box itself has a title, a key, and a value. The list items have titles and values.



Type: List Box
Title:
Key:
Default Value:
Store in secure settings
Titles:
Values:

Radio Buttons Settings

You should have at least two radio buttons for this user interface item to make sense. The user must choose one, but only one, of the radio buttons. Each radio button has a title that the user sees and a value that the button selects for.


Extension Settings
Setting Items: Setting Item 1
Type: Radio Buttons
Title:
Key:
Default Value:
Store in secure settings
Titles:
Values:

Groups and Separators

If your user settings should be grouped, enter a group label before each block of settings. This puts a large, bold heading before the group.

If you want to put separators between settings, insert a separator.

How to Use the Settings API

The display and user interface for settings are managed by Safari. You can use the `SafariExtensionSettings` class to get the current value of a setting before using it. Use in the setting's key as a property name to get its current value:

```
var myVolume = safari.extension.settings.volume
```

Whenever a setting is changed, Safari generates a `"change"` event. The event is generated whether the change is made programmatically or by a user. The target of the event is the `settings` or `secureSettings` object.

To find out which value has changed, read the `key` property of the event:

```
var mySettingKey = event.key
```

The new value and old value of the setting are in the `newValue` and `oldValue` properties of the event.

To be sure you are using the current value of your settings, you must either install an event listener for the `"change"` event in your global page or extension bar, or get the value of your variables directly from the `safari.extension.settings` property immediately before using them.

Listing 16-1 (page 104) installs a listener function for an extension with only one user setting, whose key is `"volume"`. The function updates a global variable whenever the setting changes.

Listing 16-1 Responding to settings changes

```
var myVolume
function volumeChanged {
    if (event.key == "volume") {
        myVolume = event.newValue;
    }
}

safari.extension.settings.addEventListener("change", volumeChanged, false);
```

You can set values programmatically by setting the `safari.extension.settings.`*key* property to the desired value. For example:

```
safari.extension.settings.volume = myVolume
```

or:

```
safari.extension.secureSettings.volume = myVolume
```

Using HTML5 Local Storage

Safari provides full support for HTML5 client-side storage, both simple local storage of key-value pairs, and the client-side database API that allows you to create persistent relational databases on the client machine.

When used from an injected script, the domain of the local storage is the domain of the webpage the script is injected into. In other words, local data is stored with the webpage's data.

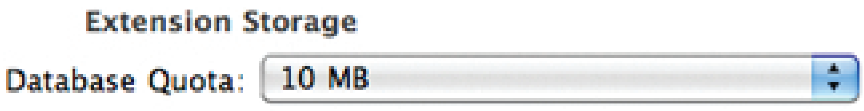
For injected scripts, the amount of database storage available is set in the user's security preferences.

In Safari 5.0.1 and later, you can use client-side databases in your global page or in extension bars as well. When used from a global HTML page or extension bar, the domain of the local storage is the extension. This data belongs to your extension.

Note: In Safari 5.1, attempting to use local storage to convey data between the global page or an extension bar and full page content does not work. As a workaround for this limitation, use message passing to convey the data. Message passing is the best mechanism for this purpose.

To use client-side databases from your global page or an extension bar, you need to allocate database storage for your extension in the Extension Storage section of Extension Builder, as shown in Figure 16-3.

Figure 16-3 Extension storage


Extension Storage
Database Quota: 10 MB

The default storage amount is none. You can choose a value from 1 to 100 megabytes.

Important: If you do not allocate storage in Extension Builder, any call to `openDatabase` from the global page or an extension bar returns `null`.

The local storage API is documented in *Safari Client-Side Storage and Offline Applications Programming Guide* .

Debugging Extensions

To test and debug extensions, use Safari’s built-in Web Inspector. Learn how to use these tools by reading *Safari Web Inspector Guide* .

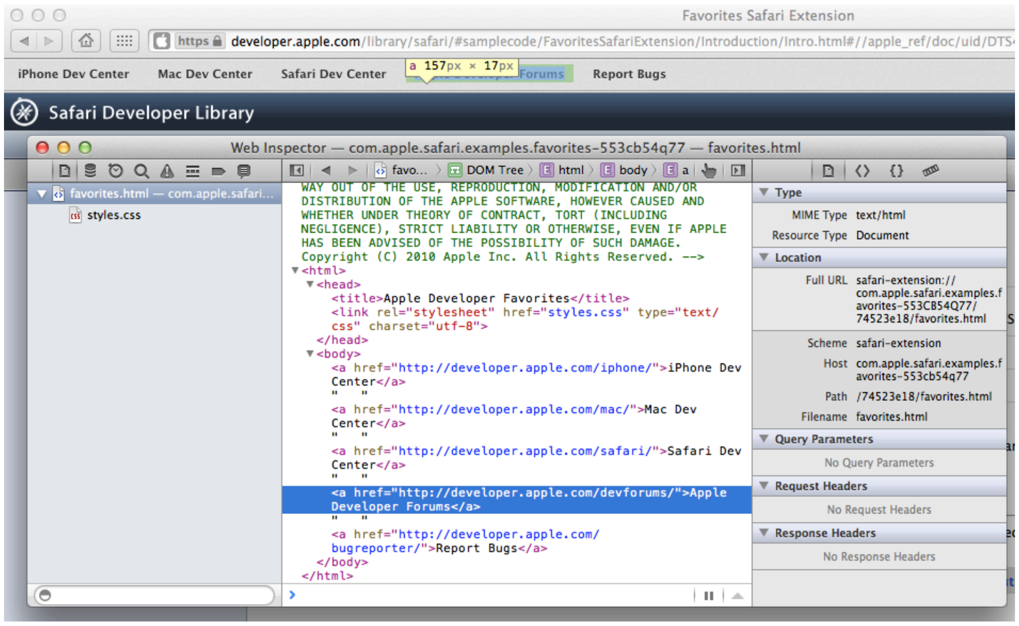
This chapter shows you how to apply the developer tools specifically to Safari extensions, but does not go into detail about using the tools.

Debugging Extension Bars

To debug an extension bar, Control-click or right-click on the extension bar. From the context menu presented, select Inspect Element.

This opens Web Inspector with the element in the extension bar selected, as illustrated in Figure 17-1.

Figure 17-1 Inspecting extension bars



Favorites Safari Extension
https developer.apple.com/library/safari/#samplecode/FavoritesSafariExtension/Introduction/Intro.html#//apple_ref/doc/uid/DTS
iPhone Dev Center
Mac Dev Center
Safari Dev Center
a 157px × 17px
Forums
Report Bugs
Safari Developer Library
Web Inspector — com.apple.safari.examples.favorites-553cb54q77 — favorites.html
favorites.html — com.apple.safari...
styles.css
favo...
DOM Tree
html
body
a

```
WAY OUT OF THE USE, REPRODUCTION, MODIFICATION AND/OR
DISTRIBUTION OF THE APPLE SOFTWARE, HOWEVER CAUSED AND
WHETHER UNDER THEORY OF CONTRACT, TORT (INCLUDING
NEGLIGENCE), STRICT LIABILITY OR OTHERWISE, EVEN IF APPLE
HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Copyright (C) 2010 Apple Inc. All Rights Reserved. -->
▼<html>
  ▼<head>
     <title>Apple Developer Favorites</title>
     <link rel="stylesheet" href="styles.css" type="text/
     css" charset="utf-8">
   </head>
  ▼<body>
     <a href="http://developer.apple.com/iphone/">iPhone Dev
     Center</a>
     "   "
     <a href="http://developer.apple.com/mac/">Mac Dev
     Center</a>
     "   "
     <a href="http://developer.apple.com/safari/">Safari Dev
     Center</a>
     "   "
     <a href="http://developer.apple.com/devforums/">Apple
     Developer Forums</a>
     "   "
     <a href="http://developer.apple.com/
     bugreporter/">Report Bugs</a>
   </body>
</html>
```

Type

MIME Type	text/html
Resource Type	Document

Location

Full URL	safari-extension://com.apple.safari.examples.favorites-553CB54Q77/74523e18/favorites.html
Scheme	safari-extension
Host	com.apple.safari.examples.favorites-553cb54q77
Path	/74523e18/favorites.html
Filename	favorites.html

Query Parameters

No Query Parameters

Request Headers

No Request Headers

Response Headers

No Response Headers

The Resource navigator shows the DOM tree for the extension bar, including any inline JavaScript. If your extension bar includes any `.js` files or other resources, they will appear under the Resource navigator. JavaScript errors are listed under the Issue navigator.

Clicking an element highlights it in the extension bar of the browser window, displays its attributes and styles, and allows you to edit them interactively (double-click an attribute or style value to edit it). If you use the Node navigator, you can display event listeners for all nodes, or any given node.

In addition, the `console` API is supported for extension bars and is compatible with Firebug. You can log data to the console and see it using the Log navigator in Web Inspector. For example, you can call the following line of JavaScript in your extension to log the URL of the current tab:

```
console.log(safari.application.activeBrowserWindow.tabs[0].url);
```

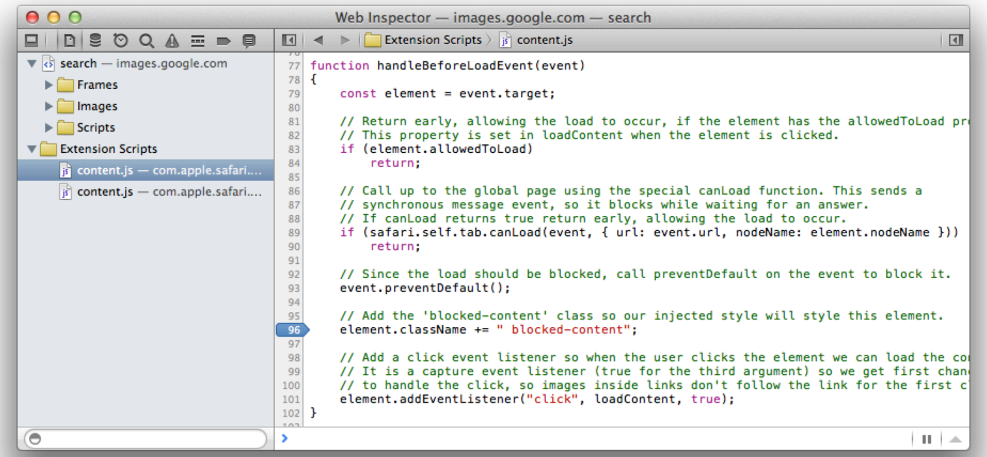
You can enable JavaScript debugging, set breakpoints, profile your JavaScript, look at how resources load, or examine HTML5 local storage databases, just as you would on a website you were developing. For details, see *Safari Web Inspector Guide* .

Debugging Injected Scripts

Debugging an injected script or style sheet with the Safari developer tools is very much like debugging a script or style sheet for your own website. The main difference is that you must load a webpage to which your extension has access in order to inject the script for debugging.

Choose Show Web Inspector from the Develop menu, or right-click on the webpage and choose Inspect Element. Then under the Resource navigator, choose your script from the folder labeled Extension Scripts.

Figure 17-2 Inspecting injected scripts


Web Inspector — images.google.com — search
Extension Scripts
content.js
search — images.google.com
Frames
Images
Scripts
Extension Scripts
content.js — com.apple.safari....
content.js — com.apple.safari....

```
function handleBeforeLoadEvent(event)
{
    const element = event.target;

    // Return early, allowing the load to occur, if the element has the allowedToLoad pr
    // This property is set in loadContent when the element is clicked.
    if (element.allowedToLoad)
        return;

    // Call up to the global page using the special canLoad function. This sends a
    // synchronous message event, so it blocks while waiting for an answer.
    // If canLoad returns true return early, allowing the load to occur.
    if (safari.self.tab.canLoad(event, { url: event.url, nodeName: element.nodeName }))
        return;

    // Since the load should be blocked, call preventDefault on the event to block it.
    event.preventDefault();

    // Add the 'blocked-content' class so our injected style will style this element.
    element.className += " blocked-content";

    // Add a click event listener so when the user clicks the element we can load the co
    // It is a capture event listener (true for the third argument) so we get first chan
    // to handle the click, so images inside links don't follow the link for the first c
    element.addEventListener("click", loadContent, true);
}
```

Any detected errors in the script are shown in the Issue navigator. Click in the gutter between the line number and script to set a breakpoint. The Firebug-compatible console API is available for logging data.

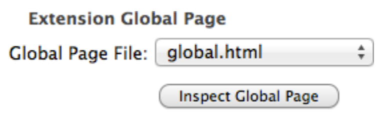
Note: Currently, you cannot use the console to interactively enter the names of injected script objects, such as variables or functions defined in the script. You can log data to the console using the console API, set breakpoints, profile scripts, and so on, but the private namespace of your injected script keeps the console from recognizing object names defined in the script.

For more information on debugging scripts, see *Safari Web Inspector Guide* .

Debugging a Global HTML Page

To bring up Web Inspector for the global page, choose Show Extension Builder from the Develop menu. When your extension has a global HTML page, a button is added to the Extension Builder interface, in the Global Page section, as illustrated in Figure 17-3.

Figure 17-3 Inspect Global Page button



Click Inspect Global Page to open Web Inspector with your global page selected for debugging.

For details on debugging HTML and JavaScript using Safari's integrated tools, see *Safari Web Inspector Guide* .

Distributing Your Extension

You are free to distribute your extension by download from your web server, on disk, by email, or using any of the usual methods to distribute digital media. You may also submit your extension for inclusion in Apple's Safari Extension Gallery.

Putting Your Extension on a Web Server

Include a link to a copy of your `.safariextz` folder on your website.

Be sure to include a description of what your extension does.

Make sure your web server is serving the extension using the MIME type `application/octet-stream`.

Most web servers maintain a table of file extensions and MIME types, and provide an administrative tool for updating the table.

For example, to add a MIME type to an Apache web server, use the AddType directive:

```
AddType application/octet-stream .safariextz
```

For IIS web servers, the MIME settings are typically accessed using the MMC by right-clicking the host computer name and choosing Properties, then adding a new MIME setting and file extension.

For more information, consult the vendor's documentation for your web server, or do a web search for "add MIME type" + YourWebServer + YourVersionNumber.

Submitting Your Extension to the Apple Gallery

Go to developer.apple.com and log in as a Safari Developer. There is a form on the website to submit your app.

Be sure to post your extension on a web server with the MIME type for `.safariextz` files set to `application/octet-stream`.

The Safari Extension Gallery includes a link to your extension. Your extension is not mounted on the Apple web server.

Updating Extensions

You can have Safari automatically check for updates to your extension and offer to download and install an update when one becomes available.

To enable automatic updates, create a text file with the `.plist` file extension and put it on a web server, then include the URL of the file in the Update Manifest field of Extension Builder.

The `.plist` file is XML with this basic structure:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

<dict>

   <key>Extension Updates</key>

   <array>

     <dict>

       <key>CFBundleIdentifier</key>

       <string>com.yourCompany.safari.yourExtensionName</string>

       <key>Developer Identifier</key>

       <string>YourCertificateID</string>

       <key>CFBundleVersion</key>

       <string>Your current bundle version</string>

       <key>CFBundleShortVersionString</key>

       <string>Your current display version</string>

       <key>URL</key>

       <string>Your-.safariextz-URL</string>

     </dict>

   </array>

</dict>

</plist>
```

Copy the structure, but replace the contents of the `<string>` elements with the data for your extension, leaving all other elements exactly as shown.

If your Developer ID is shown in Extension Builder as:

Safari Developer: (12A345BCDE) you@yourmail.com

Then `YourCertificateID` for the update manifest is: 12A345BCDE.

`Your-.safariextz-URL` must be a valid URL to download the current version of your extension. Be sure your web server has the `.safariextz` file extension associated with the MIME type `application/octet-stream`. For more information, see "Putting Your Extension on a Web Server" (page 109).

If you have more than one extension, you can maintain a single update manifest for all of them. The form for multiple extensions is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
   <key>Extension Updates</key>
   <array>
      <dict>
        <key>CFBundleIdentifier</key>
        <string>com.yourCompany.safari.firstExtensionName</string>
        ...
      </dict>
      <dict>
        <key>CFBundleIdentifier</key>
        <string>com.yourCompany.safari.nextExtensionName</string>
        ...
      </dict>
   </array>
</dict>
</plist>
```

Include one `<dict>` element inside the `<array>` element for every extension.

Document Revision History

This table describes the changes to *Safari Extensions Development Guide* .

Date	Notes
2014-03-10	Added certificate information to "Extensions Overview."
2013-09-18	Corrected the "Global HTML Page" description to remove audio element support.
2013-06-13	Added a section explaining to query the page's visibility before dispatching messages to a tab.
2012-07-23	Updated for Safari 6.0. Made minor edits throughout.
2012-02-16	Several small whitespace fixes.
2011-12-14	Added recommendations for extension security and high-resolution icons.
2011-07-07	Updated for Safari 5.1 to include popovers, Reader API, tab and window events.
2010-09-01	Updated for Safari 5.0.1. Corrected typos. Edited for clarity and consistency.
2010-08-03	Fixed typos and code errors. Expanded contextual menu section and clarified update manifest.
2010-06-21	New document.