Safari Web Content Guide

2010-12-16

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Introduction **Introduction 11**

Who Should Read This Document 12
Organization of This Document 12
See Also 13

Chapter 1 **Creating Compatible Web Content 15**

Use Standards 15
Follow Good Web Design Practices 16
Use Security Features 16
Avoid Framesets 17
Use Columns and Blocks 18
Know iOS Resource Limits 19
    Checking the Size of Webpages 20
Use the Select Element 20
Use Supported JavaScript Windows and Dialogs 21
Use Supported Content Types and iOS Features 22
Use Canvas for Vector Graphics and Animation 24
Use the HTML5 Audio and Video Elements 24
Use Supported iOS Rich Media MIME Types 24
Don't Use Unsupported iOS Technologies 25

Chapter 2 **Optimizing Web Content 29**

Using Conditional CSS 29
Using the Safari User Agent String 31

Chapter 3 **Configuring the Viewport 35**

Layout and Metrics on iPhone and iPod touch 35
What Is the Viewport? 36
    Safari on the Desktop Viewport 37
    Safari on iOS Viewport 38
    Examples of Viewports on iOS 38
Default Viewport Settings 42
Using the Viewport Meta Tag 42
Changing the Viewport Width and Height 43
How Safari Infers the Width, Height, and Initial Scale 45
Viewport Settings for Web Applications 48

Chapter 4 Customizing Style Sheets 51

Leveraging CSS3 Properties 51
Adjusting the Text Size 51
Highlighting Elements 52

Chapter 5 Designing Forms 55

Laying Out Forms 55
Customizing Form Controls 57
Configuring Automatic Correction and Capitalization 59

Chapter 6 Handling Events 61

One-Finger Events 61
Two-Finger Events 64
Form and Document Events 65
Making Elements Clickable 65
Handling Multi-Touch Events 66
Handling Gesture Events 68
Preventing Default Behavior 69
Handling Orientation Events 69
Supported Events 71

Chapter 7 Configuring Web Applications 73

Specifying a Webpage Icon for Web Clip 73
Specifying a Startup Image 74
Hiding Safari User Interface Components 74
Changing the Status Bar Appearance 75

Chapter 8 Creating Video 77

Sizing Movies Appropriately 78
Don't Let the Bit Rate Stall Your Movie 78
Using Supported Movie Standards 78
Encoding Video for Wi-Fi, 3G, and EDGE 78
Creating a Reference Movie 79
Creating a Poster Image for Movies 80
Configuring Your Server 81

Chapter 9 Storing Data on the Client 83

Creating a Manifest File 83
Declaring a Manifest File 84
Updating the Cache 84

Handling Cache Events 85

Chapter 10 **Getting Geographic Locations 87**

Geographic Location Classes 87
Getting the Current Location 87
Tracking the Current Location 88
Handling Location Errors 88

Chapter 11 **Debugging 89**

Enabling the Safari Console 89
Viewing Console Messages 91
Creating Messages in JavaScript 93

Appendix A **HTML Basics 95**

What Is HTML? 95
Basic HTML Structure 95
Creating Effective HTML Content 97
Using Other HTML Features 99

Appendix B **CSS Basics 101**

What Is CSS? 101
Inline CSS 101
Head-Embedded CSS 102
External CSS 103

Document Revision History 105

Figures, Tables, and Listings

Chapter 1 **Creating Compatible Web Content 15**

Figure 1-1 Comparison of frameset on the desktop and iOS 17
Figure 1-2 Comparison of no columns vs. columns 18
Figure 1-3 Comparison of the select element on the desktop and iOS 21
Figure 1-4 Confirm dialog 22
Figure 1-5 Playing video on iOS 23
Figure 1-6 Viewing PDF documents on iOS 23
Table 1-1 Supported iOS rich media MIME types 25

Chapter 2 **Optimizing Web Content 29**

Figure 2-1 Small device rendering 30
Figure 2-2 Desktop rendering 30
Listing 2-1 Screen-specific style sheet 31
Listing 2-2 Print-specific style sheet 31
Listing 2-3 iPhone running on iOS 2.0 user agent string 32
Listing 2-4 iPod touch running iOS 1.1.3 user agent string 32
Listing 2-5 iPad running iOS 3.2 user agent string 32
Listing 2-6 iPhone running iOS 1.0 user agent string 32

Chapter 3 **Configuring the Viewport 35**

Figure 3-1 Layout and metrics in portrait orientation 36
Figure 3-2 Differences between Safari on iOS and Safari on the desktop 37
Figure 3-3 Safari on desktop viewport 38
Figure 3-4 Viewport with default settings 39
Figure 3-5 Viewport with width set to 320 39
Figure 3-6 Viewport with width set to 320 and scale set to 150% 40
Figure 3-7 Viewport with width set to 320 and scale set to 50% 41
Figure 3-8 Viewport with arbitrary user scale 41
Figure 3-9 Default settings work well for most webpages 42
Figure 3-10 Comparison of 320 and 980 viewport widths 43
Figure 3-11 Webpage is too narrow for default settings 44
Figure 3-12 Web application page is too small for default settings 45
Figure 3-13 Default width and initial scale 46
Figure 3-14 Default width with initial scale set to 1.0 46
Figure 3-15 Width set to 320 with default initial scale 47
Figure 3-16 Width set to 200 with default initial scale 47
Figure 3-17 Width set to 980 and initial scale set to 1.0 48
Figure 3-18 Not specifying viewport properties 49
Figure 3-19 Width set to device-width pixels 49

Chapter 4 **Customizing Style Sheets 51**

Figure 4-1 Comparison of text adjustment settings 52
Figure 4-2 Differences between default and custom highlighting 53
Listing 4-1 Setting the text size adjustment property 52
Listing 4-2 Changing the tap highlight color 53

Chapter 5 **Designing Forms 55**

Figure 5-1 Form metrics when the keyboard is displayed 56
Figure 5-2 A custom checkbox 57
Figure 5-3 A custom text field 58
Figure 5-4 A custom select element 58
Table 5-1 Form metrics 56
Listing 5-1 Creating a custom checkbox with CSS 57
Listing 5-2 Creating a custom text field with CSS 58
Listing 5-3 Creating a custom select control with CSS 59

Chapter 6 **Handling Events 61**

Figure 6-1 The panning gesture 62
Figure 6-2 The touch and hold gesture 62
Figure 6-3 The double-tap gesture 63
Figure 6-4 One-finger gesture emulating a mouse 64
Figure 6-5 The pinch open gesture 64
Figure 6-6 Two-finger panning gesture 65
Table 6-1 Types of events 71
Listing 6-1 A menu using a mouseover handler 65
Listing 6-2 Adding an onclick handler 66
Listing 6-3 Displaying the orientation 70

Chapter 8 **Creating Video 77**

Figure 8-1 Export movie panel 79
Figure 8-2 Reference movie components 80
Table 8-1 File name extensions for MIME types 81

Chapter 9 **Storing Data on the Client 83**

Listing 9-1 Sample manifest file 83

Chapter 11 **Debugging 89**

Figure 11-1 Selecting Developer settings 90
Figure 11-2 Enabling the Debug Console 90

Figure 11-3 The message banner 91
Figure 11-4 Messages in the console 92
Figure 11-5 Filtered console messages 92
Figure 11-6 Viewport width or height tip 93
Figure 11-7 JavaScript timeout message 93
Figure 11-8 Console messages from your JavaScript code 94

Appendix A HTML Basics 95

Listing A-1 Basic HTML document 95
Listing A-2 Adding a paragraph 97
Listing A-3 Adding a heading 97
Listing A-4 Creating a hyperlink 97
Listing A-5 Adding an image 98
Listing A-6 Creating a table 99

Appendix B CSS Basics 101

Listing B-1 The styles.css file 103

Introduction

Safari runs on multiple operating systems and devices. All versions of Safari—Safari on the desktop and Safari on iOS—use the same WebKit engine. Therefore, web content intended for the desktop might work well on devices running iOS without any modifications. Some differences exist, however, so at a minimum you should ensure that your webpages are compatible with Safari on iOS. Next, you might optimize your webpages for iOS simply as a convenience to the user. For example, ensure that your webpages work over Wi-Fi, 3G, and EDGE, scale correctly when rendered, and contain media that is viewable on iOS. There are also a few modifications you can make for specific devices such as iPad. Finally, you might build custom web applications for either platform that look and behave like native applications.

Safari on the desktop is the Safari application that runs on Mac OS X and Windows. It is a full-featured web browser for the desktop that supports industry standards as well as many WebKit extensions. In addition, it includes a number of tools that developers can use to analyze, test, and debug websites and web applications.

Safari on iOS, the application for browsing the web on devices such as iPhone, iPod touch, and iPad, is also a full web browser running on a small handheld device with a high-resolution screen. This unique implementation of Safari responds to a finger as the input device and supports gestures for zooming and panning. It also renders webpages in portrait or landscape orientation. It contains many built-in features such as PDF viewing, video playback, and support for links to the native Phone, Mail, Maps, and YouTube applications.

The **WebKit** is an open source project as well as a framework in Mac OS X that lets developers embed a web browser in their Cocoa applications. WebKit has a JavaScript and Objective-C interface to access the Document Object Model (DOM) of a webpage. Dashboard, Mail, and many other Mac OS X applications also use WebKit as an embedded browser. You can use the `UIWebView` class in UIKit on iOS to embed a web browser in an native application.

In addition to providing browser functionality, WebKit also implements some extensions to HTML, CSS, and JavaScript, including several specific to Safari. Safari extensions include CSS animation and transform properties, and JavaScript database support. Safari on iOS includes JavaScript multitouch event support. Some extensions are fully supported on both platforms and others are not. Check reference documents for specific availability of those features you want to use.

This document teaches you how to create web content compatible with Safari running on any platform and how to separate your iOS-specific web content from your other web content so that when you optimize your web content for iOS, it still works on the desktop and other browsers. This document also covers some basics on tailoring web applications for iOS.

iOS Note: Safari on iOS behaves the same on different devices except when the user taps links to device-only applications. Read *Apple URL Scheme Reference* for information on the links that behave differently on other devices. On iPad, the behavior of HTML5 media elements is similar to the desktop. Read *Safari HTML5 Audio and Video Guide* for the differences on iPad.

Who Should Read This Document

You should read this document if you want your web content to look good and perform well on either the desktop or iOS, plan to write iOS-specific web content, use iOS-specific style sheets, or use iOS native application links in your web content. Definitely read this document if you are creating a custom web application for either platform.

iOS Note: Also read *iOS Human Interface Guidelines* which describes how Safari on iOS behaves and contains metrics and tips on designing user interfaces for iOS. Understanding how Safari on iOS presents web content to the user and how the user can zoom, pan, and double-tap on your webpages are prerequisites for reading this document.

Organization of This Document

This chapter covers important information that you should read first:

- "Creating Compatible Web Content" (page 15) provides guidelines for creating web content that is compatible with Safari on the desktop and Safari on iOS.

This chapter covers the first steps you need to follow to optimize web content for Safari:

- "Optimizing Web Content" (page 29) describes how to detect Safari on different platforms and use conditional Cascading Style Sheets (CSS) so that you can begin optimizing web content for iOS.

These chapters describe different ways to optimize web content for iOS:

- "Configuring the Viewport" (page 35) explains how to use the viewport tag to control the layout of your webpages.
- "Customizing Style Sheets" (page 51) covers how to adjust the text size when zooming and how to control highlighting using CSS.
- "Designing Forms" (page 55) explains how to lay out forms, design custom form controls, and turn auto correction and capitalization on and off.
- "Handling Events" (page 61) provides information on what events you can handle in JavaScript.

This chapter describes how to create video content for multiple platforms:

- "Creating Video" (page 77) explains how to create video content for playback on iOS in general, including video content embedded in your webpages.

This chapter covers information on how to store data locally:

- "Storing Data on the Client" (page 83) describes how to use the HTML5 application cache for storing resources locally.

This chapter covers information on how to debug web content:

- "Debugging" (page 89) describes the Safari on iOS console which you use to help test and debug your webpages.

If you are new to web development, read these appendixes that provide introductions to HTML and CSS:

- "HTML Basics" (page 95) provides an overview on how to create structured HTML web content.
- "CSS Basics" (page 101) describes how to add style sheets to existing HTML web content.

See Also

There are a variety of other resources for Safari web content developers in the ADC Reference Library.

If you are creating an iOS web application, then you should also read:

- *iOS Human Interface Guidelines*, which provides user interface guidelines for designing webpages and web applications for Safari on iOS.
- *Apple URL Scheme Reference*, which describes how to use the Phone, Mail, Text, YouTube, iTunes, and Maps applications from your webpages.

If you want to learn more about visual effects, then you should read:

- *Safari CSS Visual Effects Guide*, which describes how to use the CSS visual effects properties including the transition, animation, and transforms properties. It also covers the JavaScript APIs for handling visual effects events.

If you want to embed video and audio in your webpages read:

- *Safari HTML5 Audio and Video Guide*, which describes how to use the HTML5 audio and video elements in your webpages.

If you want to learn more about JavaScript multi-touch event support, then you should read:

- *Safari DOM Additions Reference*, which describes DOM extensions including touch event classes that you use to handle multi-touch gestures in JavaScript on iOS.

If you want to learn more about JavaScript database support, then you should read:

- *Safari Client-Side Storage and Offline Applications Programming Guide*, which describes a simple relational database that you can use to store persistent data in JavaScript that cannot be stored in cookies.

If you want to use the JavaScript media APIs, then you should read:

- *JavaScript Scripting Guide for QuickTime*, which describes how to use JavaScript to query and control the QuickTime plug-in directly.

If you want to learn more about what HyperText Markup Language (HTML) tags and Cascading Style Sheets (CSS) properties are supported in Safari, then read:

- *Safari HTML Reference*, which describes the HTML elements and attributes supported by various Safari and WebKit applications.
- *Safari CSS Reference*, which describes the CSS properties supported by various Safari and WebKit applications.

If you are using JavaScript and want access to the DOM or use the canvas object, then read:

- *WebKit DOM Programming Topics*, which describes how to use JavaScript in web content for WebKit-based applications.
- *WebKit DOM Reference*, which describes the API for accessing WebKit's Document Object Model.

If you are developing web content for Safari on the desktop and iOS, then you should read:

- *Safari User Guide for Web Developers*, which describes how to use the Debug menu in Safari.
- *Dashcode User Guide*, which describes how to use Dashcode to create web applications.

If you want to embed a browser in your iOS application, then read:

- *UIWebView Class Reference* for a description of the `UIWebView` class.

If you want to learn more about WebKit or contribute to the open source project, then go to The WebKit Open Source Project.

If you want to read the WebKit W3C proposals, then go to: http://www.webkit.org/specs.

Creating Compatible Web Content

This chapter covers best practices in creating web content that is compatible with Safari on the desktop and Safari on iOS. Many of these guidelines simply improve the reliability, performance, look, and user experience of your webpages on both platforms. If your target is iOS, the first step is to get your web content working well on the desktop. If your target is the desktop, with minimal modifications, you can get your web content to look good and perform well on iOS too.

For example, you need to pay attention to the layout of your content and execution limits on iOS. If you use conditional CSS, as recommended in "Optimizing Web Content" (page 29), your webpages optimized for iOS still work in other browsers. Read the rest of this document for how to optimize your web content for Safari.

iOS Note: When designing your webpages, be aware of how Safari on iOS presents webpages to the user and how the user interacts with your webpages using gestures to zoom, pan, and double-tap. Read *iOS Human Interface Guidelines* for metrics and tips on designing user interfaces for iOS.

Use Standards

The first design rule is to use web standards. Standards-based web development techniques ensure the most consistent presentation and functionality across all modern browsers, including Safari. A well-designed website probably requires just a few refinements to look good and work well on Safari.

The WebKit engine, shared by Safari on the desktop and Safari on iOS, supports all the latest modern web standards, including:

- HTML 4.01
- XHTML 1.0
- CSS 2.1 and partial CSS3
- ECMAScript 3 (JavaScript)
- DOM Level 2
- AJAX technologies, including XMLHttpRequest

The web is always evolving, and as it does, so does WebKit and Safari. You'll want to keep informed of the evolving standards emanating from the Web Hypertext Application Technology Working Group (WHATWG) and World Wide Web Consortium (W3C) standards bodies. The WHATWG and W3C websites are a good place to start learning more about these standards and the upcoming HTML5:

www.whatwg.org
www.w3.org

Refer to Safari reference documents, such as *Safari HTML Reference* and *Safari CSS Reference*, for availability of features on specific platforms.

Follow Good Web Design Practices

You should follow well-established rules of good web design. This section covers a few basic rules that are critical for Safari. Read Web Page Development: Best Practices for more general advice on designing webpages.

- Add a `DOCTYPE` declaration to your HTML files.

  Preface your HTML files with a `DOCTYPE` declaration, which tells browsers which specification to parse your webpage against. See "HTML Basics" (page 95) for how to do this.

- Separate your HTML, CSS, and JavaScript into different files.

  Your webpages are more maintainable if you separate page content into distinct files for mark-up, presentation, and interaction.

- Use well-structured HTML.

  You increase cross-platform browser compatibility by running your HTML files through a validator. You should fix common problems such as missing quotes, missing close tags, incorrect nesting, incorrect case, and malformed doctype. See http://validator.w3.org or use the validator provided by your web development tools.

- Be browser independent.

  Avoid using the user agent string to check which browser is currently running. Instead, read Object Detection to learn how to determine if a browser supports a particular object, property, or method, and read Detecting WebKit with JavaScript to learn how to detect specific WebKit versions. Also use the W3C standard way of accessing page objects—that is, use `getElementByID("elementName")`. Only as a last resort, use the user agent string as described in "Using the Safari User Agent String" (page 31) to detect Safari on iOS.

Read "HTML Basics" (page 95) and "CSS Basics" (page 101) for how to write structured HTML and add CSS to existing HTML.

Use Security Features

Safari on all platforms uses the same SSL implementation to provide end-to-end security. The same encryption that prevents listening on the wire is just as secure when used in a wireless situation, whether through Wi-Fi, 3G, or EDGE. Specifically, Safari supports:

- SSL 2, SSL 3, and TLS with many popular cipher suites
- RSA keys up to 4096
- HTTPS

iOS Note: Note that the Diffie-Hellman protocol, DSA keys, and self-signed certificates are not available on iOS.

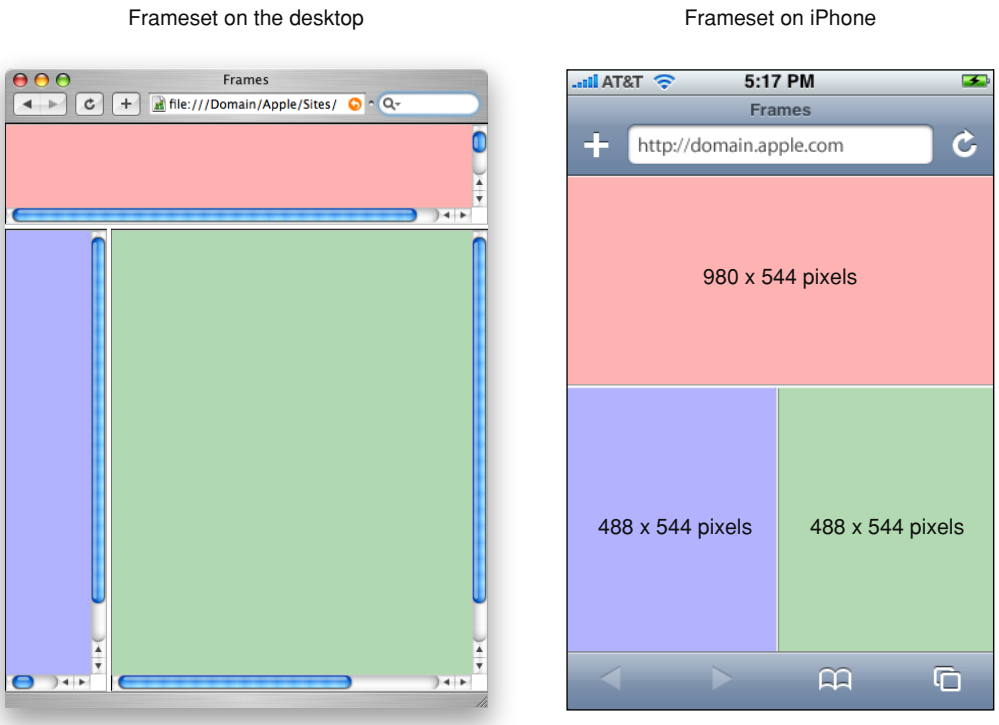
Avoid Framesets

In general, avoid using complicated framesets that rely on the ability to scroll individual frames because there are no scroll bars on iOS.

On the desktop, frames in a frameset can be independently scrolled as shown on the left in Figure 1-1. On iOS, scrollable frames in a frameset are expanded to fit their content and then a frame is scaled down to fit its region as shown on the right in Figure 1-1. Scrollable full-width inline frames are expanded to fit their content, too. All other scrollable inline frames can be panned using the two-finger gesture. See "Two-Finger Events" (page 64) for the events generated from the two-finger gesture.

Because there are no scroll bars on the inline frames, this is not an optimal user experience for viewing web content on iOS, so avoid using framesets. Instead use columns as described in "Use Columns and Blocks" (page 18).

Figure 1-1 Comparison of frameset on the desktop and iOS

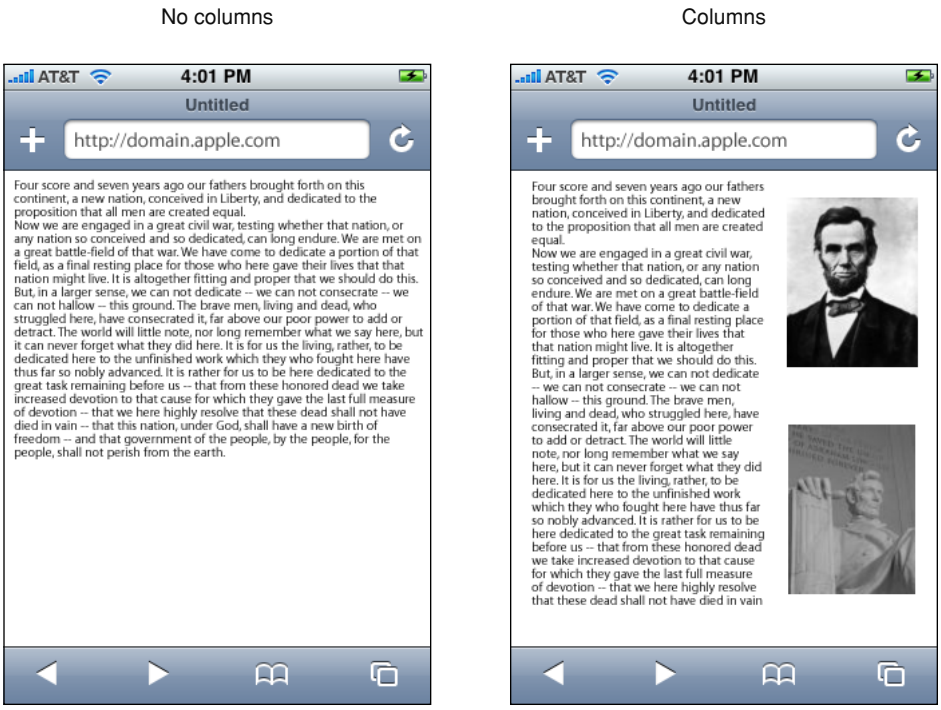


Use Columns and Blocks

To be compatible with iOS, use columns and blocks to lay out your webpage like many online newspapers. This makes your webpage more readable and also works better with double-tapping on iOS.

Text blocks that span the full width of the webpage are difficult to read on iOS as shown on the left in Figure 1-2. Columns not only break up the webpage, making it easy to read, as shown on the right in Figure 1-2, but allow the user to easily double-tap objects on the page.

Figure 1-2 Comparison of no columns vs. columns



When the user double-taps a webpage, Safari on iOS looks at the element that was double-tapped, and finds the closest block (as identified by elements like `<div>`, `<ol>`, `<ul>`, and `<table>`) or image element. If the found element is a block, Safari on iOS zooms the content to fit the screen width and then centers it. If it is an image, Safari on iOS zooms to fit the image and then centers it. If the block or image is already zoomed in, Safari on iOS zooms out.

Your webpage works well with double-tapping if you use columns and blocks. Read "CSS Basics" (page 101) for how to add CSS to existing HTML.

Know iOS Resource Limits

Your webpage performing well on the desktop is no guarantee that it will perform well on iOS. Keep in mind that iOS uses EDGE (lower bandwidth, higher latency), 3G (higher bandwidth, higher latency), and Wi-Fi (higher bandwidth, lower latency) to connect to the Internet. Therefore, you need to minimize the size of your webpage. Including unused or unnecessary images, CSS, and JavaScript in your webpages adversely affects your site's performance on iOS.

Because of the memory available on iOS, there are limits on the number of resources it can process:

- The maximum size for decoded GIF, PNG, and TIFF images is 3 megapixels.

  That is, ensure that `width * height ≤ 3 * 1024 * 1024`. Note that the decoded size is far larger than the encoded size of an image.

- The maximum decoded image size for JPEG is 32 megapixels using subsampling.

  JPEG images can be up to 32 megapixels due to subsampling, which allows JPEG images to decode to a size that has one sixteenth the number of pixels. JPEG images larger than 2 megapixels are subsampled—that is, decoded to a reduced size. JPEG subsampling allows the user to view images from the latest digital cameras.

- The maximum size for a canvas element is 3 megapixels.

  The height and width of a canvas object is 150 x 300 pixels if not specified.

- Individual resource files must be less than 10 MB.

  This limit applies to HTML, CSS, JavaScript, or nonstreamed media.

- JavaScript execution time is limited to 10 seconds for each top-level entry point.

  If your script executes for more than 10 seconds, Safari on iOS stops executing the script at a random place in your code, so unintended consequences may result.

  This limit is imposed because JavaScript execution may cause the main thread to block, so when scripts are running, the user is not able to interact with the webpage.

  Read "Debugging" (page 89) for how to debug JavaScript on iOS.

- The maximum number of documents that can be open at once is eight.

iOS Note: In iOS 1.1.4 and earlier, the JavaScript execution time was limited to 5 seconds and the size of allocations to 10 MB. Also, the limit on the size of canvas elements was the same as Safari on the desktop.

iOS Note: In iOS 2.2.1 and earlier, the sum of all of the frames needs to be less than 2 megapixels—that is, `width * height * number of frames ≤ 2 * 1024 * 1024`. In iOS 3.0 and later, the limit only applies to one frame at a time.

You also need to size images appropriately. Don't rely on browser scaling. For example, don't put a 100 x 100 image in a 10 x 10 `<img>` element. Tile small backgrounds images; don't use large background images.

Checking the Size of Webpages

You can check the size of your webpages by using Safari's Web Inspector as described in "Optimizing Download Time" in *Safari User Guide for Web Developers* or by saving your webpage as a web archive. The total size of the web archive is the size of the page and its associated resources. Follow these steps to create a web archive:

1. Choose File > Save As.
2. Enter the filename in the Save As text field.
3. Choose Web Archive from the Format pop-up menu.
4. Click Save.

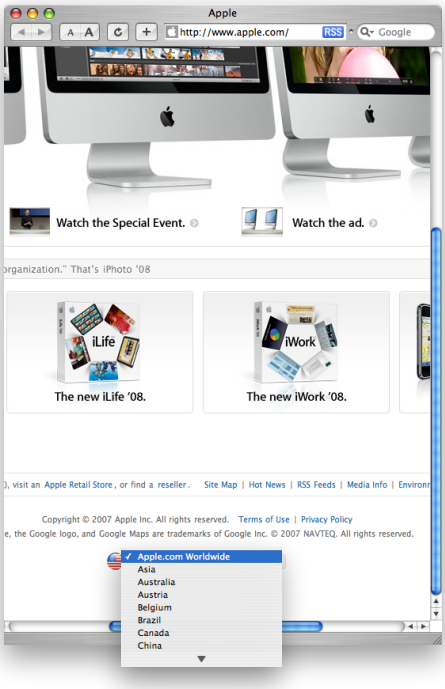
On Mac OS X, check the size of the web archive using either Finder or Terminal. Typically, pages under 30 MB work fine on iOS.

Use the Select Element

If you use the select HTML element in your webpage, iOS displays a custom select control that is optimized for selecting items in the list using a finger as the input device. On iOS, the user can flick to scroll the list and tap to select an item from the list. Figure 1-3 compares the select element on the desktop with the select element on iOS.

Figure 1-3 Comparison of the select element on the desktop and iOS

Select on the desktop



Select on iPhone

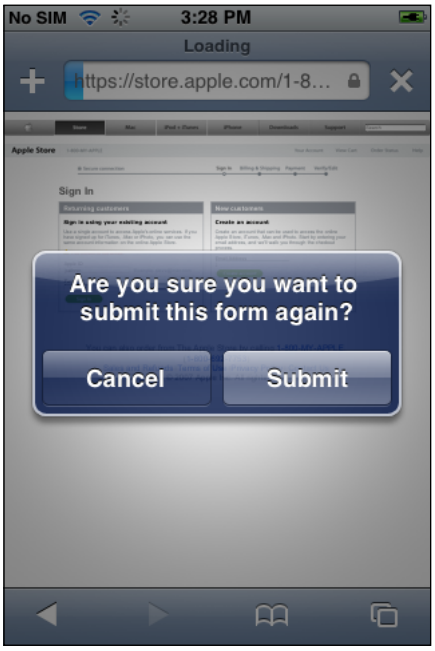
Use Supported JavaScript Windows and Dialogs

Use windows and dialogs supported by Safari on iOS and avoid the others.

You can open a new window in JavaScript by invoking `window.open()`. Remember that the maximum number of documents—hence, the maximum number of open windows—is eight on iOS.

Supported JavaScript dialog methods include `alert`, `confirm`, and `prompt`. If you use these methods, Safari on iOS displays an attractive dialog that doesn't obscure the webpage, as show in Figure 1-4.

Figure 1-4 Confirm dialog



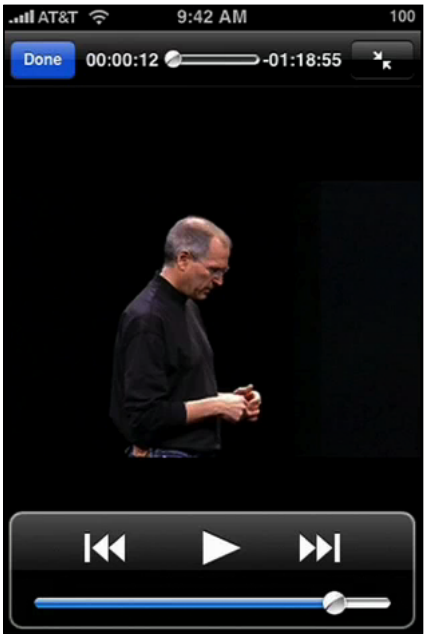
iOS Note: Note that the `showModalDialog` and `print` methods are not supported in Safari on iOS.

Use Supported Content Types and iOS Features

Be aware of the features you get for free in Safari on iOS by using supported content types and elements that tailor the presentation of content for small handheld devices with touch screens. In particular, Safari on iOS handles content types such as video and PDF files different from the desktop. Safari on iOS also has the ability to preview content types and launch another application if it is available to display that type of document. Following links such as phone numbers in your web content may launch applications too.

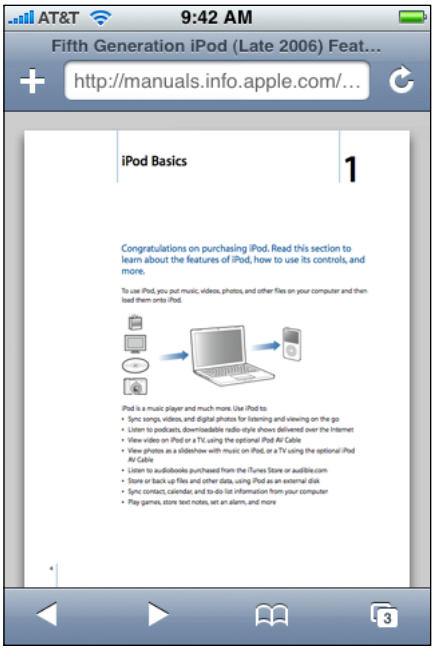
On iPhone and iPod touch, the video and audio is played back in fullscreen mode only. The video automatically expands to the size of the screen and rotates when the user changes orientation, as shown in Figure 1-5. The controls automatically hide when they are not in use. On iPad, the video and audio is played either inline in the webpage or in fullscreen mode. Read "Creating Video" (page 77) for how to export video for iOS.

Figure 1-5 Playing video on iOS



PDF documents are easy to view using Safari on iOS and even easier to page through as shown in Figure 1-6. PDF documents linked from web content are opened automatically. The page indicator keeps track of where the user is in a document. And just as with video, the user can rotate iOS to view a PDF in landscape orientation.

Figure 1-6 Viewing PDF documents on iOS



Safari on iOS previews other content types like MS Office (Word, Excel and PowerPoint), iWork (Pages, Numbers, and Keynote), and RTF documents. If another application registers for a content type that Safari on iOS previews, then that application is used to open the document. For example, on iPad, Pages may be used to open Word and Pages documents that are previewed in Safari on iOS. If another application registers for a content type that Safari on iOS doesn't support natively or preview, then Safari on iOS allows the document to be downloaded and opened using that application.

iOS Note: Previews of RTF documents is available in iOS 3.2 and later. The ability to open a downloaded file is available in iOS 3.2 and later.

When the user taps certain types of links, Safari on iOS may launch a native application to handle the link—for example, Mail to compose an email message, Maps to get directions, and YouTube to view a video. If the user taps a telephone number link on a phone device, a dialog appears asking whether the user wants to dial that number. On the desktop, most of these links redirect to the respective website. Read *Apple URL Scheme Reference* to learn more about using these types of links in your web content.

iOS Note: Note that Java and Flash content types are not supported. See "Don't Use Unsupported iOS Technologies" (page 25) for a complete list of unsupported technologies.

Use Canvas for Vector Graphics and Animation

You can use the same canvas object used by Dashboard widgets to implement sophisticated user interfaces for web applications. The canvas object was introduced in Safari 2.0, is adopted by other browser engines, and is part of the WHATWG specification. Read *WebKit DOM Programming Topics* to learn more about using the canvas object.

Use the HTML5 Audio and Video Elements

You can use the HTML5 `audio` and `video` elements to add audio and video to your webpages. On smaller devices like iPhone and iPad touch, the movie plays in full screen mode only and automatic playback is disabled so a user action is required to initiate playback. On iPad, the video plays inline in the webpage. When the video is played inline, you can create custom controls and receive media events—for example, pause and play events—to enhance the user experience. Use the `HTMLMediaElement` class and its subclasses, described in *Safari DOM Additions Reference*, to do this. Read *Safari HTML5 Audio and Video Guide* for more in-depth information on the `audio` and `video` elements. Read "Creating Video" (page 77) for how to create media files compatible with Safari.

Use Supported iOS Rich Media MIME Types

Table 1-1 lists the rich media MIME types supported by Safari on iOS. Files with these MIME types and filename extensions can be played on iOS.

Table 1-1 Supported iOS rich media MIME types

MIME type	Description	Extensions
audio/3gpp	3GPP media	3gp, 3gpp
audio/3gpp2	3GPP2 media	3g2, 3gp2
audio/aiff audio/x-aiff	AIFF audio	aiff, aif, aifc, cdda
audio/amr	AMR audio	amr
audio/mp3 audio/mpeg3 audio/x-mp3 audio/x-mpeg3	MP3 audio	mp3, swa
audio/mp4	MPEG-4 media	mp4
audio/mpeg audio/x-mpeg	MPEG audio	mpeg, mpg, mp3, swa
audio/wav audio/x-wav	WAVE audio	wav, bwf
audio/x-m4a	AAC audio	m4a
audio/x-m4b	AAC audio book	m4b
audio/x-m4p	AAC audio (protected)	m4p
video/3gpp	3GPP media	3gp, 3gpp
video/3gpp2	3GPP2 media	3g2, 3gp2
video/mp4	MPEG-4 media	mp4
video/quicktime	QuickTime Movie	mov, qt, mqv
video/x-m4v	Video	m4v

Don't Use Unsupported iOS Technologies

In general, Safari on iOS does not support any third-party plug-ins or features that require access to the file system. The following web technologies are not supported on iOS:

- Modal dialogs

  Don't use `window.showModalDialog()` or `window.print()` in JavaScript. Read "Use Supported JavaScript Windows and Dialogs" (page 21) for a list of supported dialogs.

- Mouse-over events

  The user cannot “mouse-over” a nonclickable element on iOS. The element must be clickable for a `mouseover` event to occur as described in “One-Finger Events” (page 61).

- Hover styles

  Since a `mouseover` event is sent only before a `mousedown` event, hover styles are displayed only if the user touches and holds a clickable element with a hover style. Read “Handling Events” (page 61) for all the events generated by gestures on iOS.

- Tooltips

  Similar to hover styles, tooltips are not displayed unless the user touches and holds a clickable element with a tooltip.

- Java applets

- Flash

  Don’t bring up JavaScript alerts that ask users to download Flash.

- QuickTime VR (QTVR) movies

- Plug-in installation

- Custom x.509 certificates

- WML

  Safari on iOS is not a miniature web browser—it is a full web browser that renders pages as designed—therefore, there is no need for Safari on iOS to support Wireless Markup Language (WML). Alternatively, it does support XHTML mobile profile document types and sites at `.mobi` domains.

  The XHTML mobile document type is:

  ```
  <!DOCTYPE html PUBLIC "-//WAPFORUM//DTD XHTML Mobile 1.1//EN"
  "http://www.openmobilealliance.org/tech/DTD/xhtml-mobile11.dtd">
  ```

- File uploads and downloads

  Safari on iOS does not support file uploading, that is, `<input type="file">` elements. If your webpage includes an input-file control, Safari on iOS disables it.

  Because iOS does not support file downloads, do not prompt the user to download plug-ins like Flash on iOS. See “Using the Safari User Agent String” (page 31) for how to detect Safari on iOS.

- HTML `contenteditable` Attribute

  Safari on iOS does not support editing elements on the fly. If you're using `contenteditable` to enable text input within a styled element—for example, using `<p contenteditable>` or `<div contenteditable>`)—you can replace this styled element with a styled `textarea` element. In Safari, you can customize the appearance of `textarea` elements using CSS. If necessary, you can even disable any platform-specific, built-in styling on a `textarea` element by setting `-webkit-appearance` to `none`.

By default, Safari on iOS blocks pop-up windows. However, it is a preference that the user can change. To change the Safari settings, tap Settings followed by Safari. The Block Pop-ups setting appears in the Security section.

iOS Note: Downloadable web fonts are supported in iOS 1.1.4 and earlier, and iOS 4.2 and later. SVG is supported in iOS 2.1 and later. XSLT is supported in iOS 2.0 and later.

Optimizing Web Content

The first step in optimizing web content for iOS is to separate your iOS-specific content from your desktop content and the next step is to tailor the web content for iOS. You might want to follow these steps even if iOS is not your target platform so your web content is more maintainable in the future.

Use conditional CSS so that you can create iOS-specific style sheets as described in "Using Conditional CSS" (page 29). You can also use object detection and WebKit detection as described in "Follow Good Web Design Practices" (page 16) to use extensions but remain browser-independent. Only if necessary, use the user agent string as described in "Using the Safari User Agent String" (page 31) to detect Safari on iOS or a specific device.

After optimizing your content, read the rest of the chapters in this document to learn how to set viewport properties, adjust text size, lay out forms, handle events, use application links, and export media for iOS. Finally read "Debugging" (page 89) for how to debug your webpages.

Using Conditional CSS

Once you use CSS to lay out your webpage in columns, you can use conditional CSS to create different layouts for specific platforms and mobile devices. Using CSS3 media queries, you can add iOS-specific style sheets to your webpage without affecting how your webpages are rendered on other platforms.

For example, Figure 2-1 shows a webpage containing conditional CSS specifically for iOS. Figure 2-2 shows the same webpage rendered on the desktop.

Figure 2-1 Small device rendering

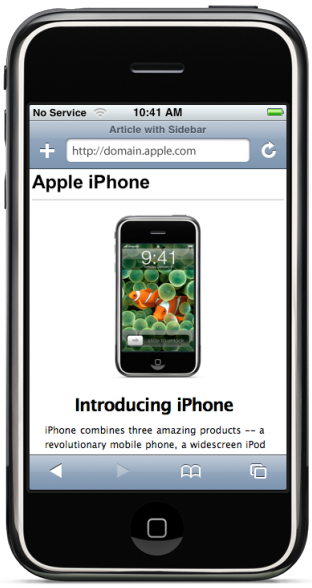
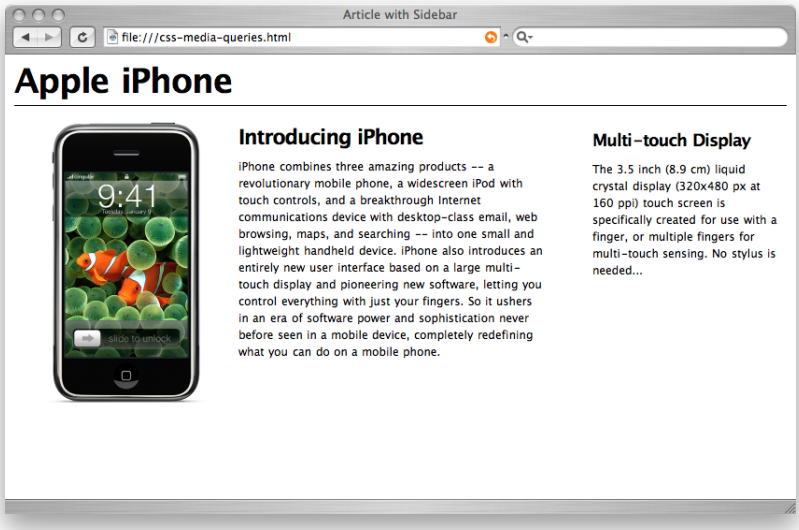


Figure 2-2 Desktop rendering



CSS3 recognizes several media types, including print, handheld, and screen. iOS ignores print and handheld media queries because these types do not supply high-end web content. Therefore, use the screen media type query for iOS.

To specify a style sheet that is just for iOS without affecting other devices, use the `only` keyword in combination with the `screen` keyword in your HTML file. Older browsers ignore the `only` keyword and won't read your iOS style sheet. Use `device-width`, `max-device-width`, and `min-device-width` to describe the screen size.

For example, to specify a style sheet for iPhone and iPod touch, use an expression similar to the following:

```
<link media="only screen and (max-device-width: 480px)" href="small-device.css"
 type= "text/css" rel="stylesheet">
```

To specify a style sheet for devices other than iOS, use an expression similar to the following:

```
<link media="screen and (min-device-width: 481px)" href="not-small-device.css"
 type="text/css" rel="stylesheet">
```

Alternatively, you can use this format inside a CSS block in an HTML file, or in an external CSS file:

```
@media screen and (min-device-width: 481px) { ... }
```

Here are some examples of CSS3 media-specific style sheets where you might provide a different style for screen and print. Listing 2-1 displays white text on dark gray background for the screen. Listing 2-2 displays black text on white background and hides navigation for print.

Listing 2-1 Screen-specific style sheet

```
@media screen {
        #text { color: white; background-color: black; }
}
```

Listing 2-2 Print-specific style sheet

```
@media print {
        #text { color: black; background-color: white; }
        #nav  { display: none; }
}
```

For more information on media queries, see: http://www.w3.org/TR/css3-mediaqueries/.

Using the Safari User Agent String

A browser sends a special string, called a **user agent**, to websites to identify itself. The web server, or JavaScript in the downloaded webpage, detects the client's identity and can modify its behavior accordingly. In the simplest case, the user agent string includes an application name—for example, `Navigator` as the application name and `6.0` as the version. Safari on the desktop and Safari on iOS have their own user agent strings, too.

The Safari user agent string for iOS is similar to the user agent string for Safari on the desktop except for two additions: It contains a platform name and the mobile version number. The device name is contained in the platform name. For example, you can detect iOS and the specific device such as iPad. Typically, you do not send iPhone-specific web content to an iPad since it has a much larger screen. Note that the version numbers in this string are subject to change over time as new versions of iOS become available, so any code that checks the user agent string should not rely on version numbers.

For example, Listing 2-3 shows the user agent string for an iPhone running iOS 2.0 and later, where the string `XXXX` is replaced with the build number.

Listing 2-3 iPhone running on iOS 2.0 user agent string

```
Mozilla/5.0 (iPhone; U; CPU iOS 2_0 like Mac OS X; en-us) AppleWebKit/525.18.1
 (KHTML, like Gecko) Version/3.1.1 Mobile/XXXXX Safari/525.20
```

The parts of the Safari on iOS user agent string are as follows:

`(iPhone; U; CPU iOS 2_0 like Mac OS X; en-us)`
: The platform string. `iPhone` is replaced with `iPod` when running on an iPod touch and `iPad` when running on an iPad.

`AppleWebKit/525.18.1`
: The WebKit engine build number.

`Version/3.1.1`
: The Safari family version.

`Mobile/XXXXX`
: The mobile version number, where `XXXX` is the build number.

`Safari/525.20`
: The Safari build number.

For example, the user agent string for an iPod touch contains `iPod` in the platform name as shown in Listing 2-4.

Listing 2-4 iPod touch running iOS 1.1.3 user agent string

```
Mozilla/5.0 (iPod; U; CPU like Mac OS X; en) AppleWebKit/420.1 (KHTML, like
Gecko) Version/3.0 Mobile/4A93 Safari/419.3
```

The user agent string for an iPad contains `iPad` in the platform name as shown in Listing 2-5.

Listing 2-5 iPad running iOS 3.2 user agent string

```
Mozilla/5.0 (iPad; U; CPU OS 3_2 like Mac OS X; en-us) AppleWebKit/531.21.10
(KHTML, like Gecko) Version/4.0.4 Mobile/7B334b Safari/531.21.10
```

Note that the user agent string is slightly different for earlier Safari on iOS releases. Listing 2-6 shows the user agent string for an iPhone running iOS 1.1.4 and earlier. Note that the platform string does not contain the iOS version number.

Listing 2-6 iPhone running iOS 1.0 user agent string

```
Mozilla/5.0 (iPhone; U; CPU like Mac OS X; en) AppleWebKit/420+ (KHTML, like
Gecko) Version/3.0 Mobile/1A543 Safari/419.3
```

Typically, you use the WebKit build number to test for supported WebKit HTML tags and CSS properties. The Safari family version, or marketing version, is included in the user agent string for Safari on the desktop, too. Therefore, you can use it to track usage statistics across all Safari platforms.

Go to these websites to learn more about other recommended techniques for detecting Safari and WebKit:

- webkit.org

  http://trac.webkit.org/projects/webkit/wiki/DetectingWebKit

  Contains JavaScript sample code for detecting Safari on iPhone and iPod touch.

- developer.apple.com

  http://developer.apple.com/internet/webcontent/objectdetection.html

Configuring the Viewport

Safari on iOS displays webpages at a scale that works for most web content originally designed for the desktop. If these default settings don't work for your webpages, it is highly recommended that you change the settings by configuring the viewport. You especially need to configure the viewport if you are designing webpages specifically for iOS. Configuring the viewport is easy—just add one line of HTML to your webpage—but understanding how viewport properties affect the presentation of your webpages on iOS is more complex. Before configuring the viewport, you need a deeper understanding of what the visible area and viewport are on iOS.

If you are already familiar with the viewport on iOS, read "Using the Viewport Meta Tag" (page 42) for details on the viewport tag and "Viewport Settings for Web Applications" (page 48) for web application tips. Otherwise, read the sections in this chapter in the following order:

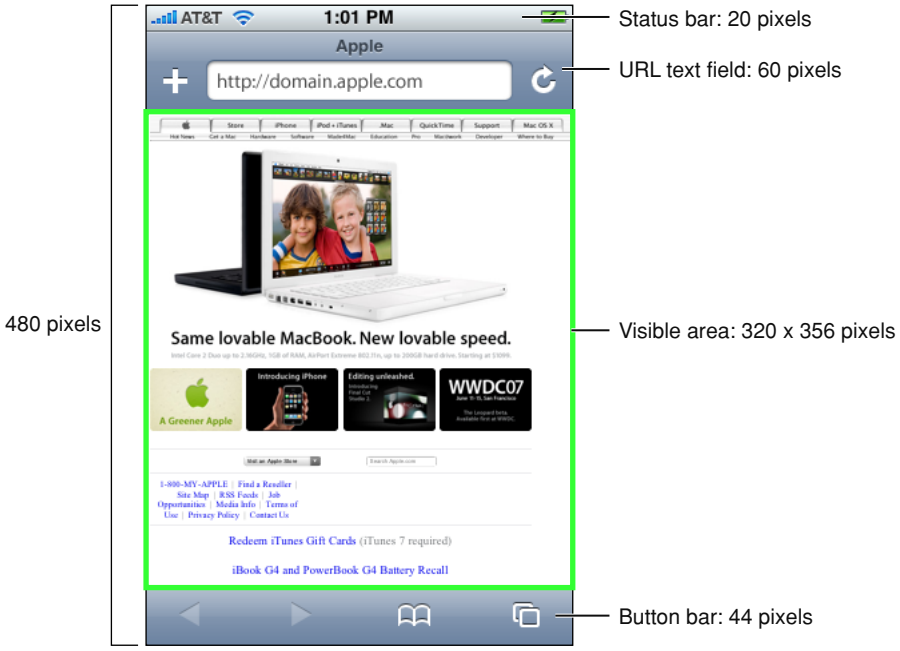
- Read "Layout and Metrics on iPhone and iPod touch" (page 35) to learn about the available screen space for webpages on small devices.
- Read "What Is the Viewport?" (page 36) for a deeper understanding of the viewport on iOS.
- Read "Default Viewport Settings" (page 42) and "Using the Viewport Meta Tag" (page 42) for how to use the viewport meta tag.
- Read "Changing the Viewport Width and Height" (page 43) and "How Safari Infers the Width, Height, and Initial Scale" (page 45) to understand better how setting viewport properties affects the way webpages are rendered on iOS.
- Read "Viewport Settings for Web Applications" (page 48) if you are designing a web application for iOS.

See "Supported Meta Tags" for a complete description of the viewport meta tag.

Layout and Metrics on iPhone and iPod touch

Because Safari on iOS adds controls above and below your web content, you don't have access to the entire screen real estate. In portrait orientation, the visible area for web content on iPhone and iPod touch is 320 x 356 pixels as shown in Figure 1-1. In landscape orientation, the visible area is 480 x 208 pixels.

Figure 3-1 Layout and metrics in portrait orientation



Note that if the URL text field is not in use, it is anchored above the webpage and moves with the webpage when the user pans. This adds 60 pixels to the height of the visible area. However, since the URL text field can appear at any time, you should not rely on this extra real estate when designing your webpage. Video playback uses the entire screen on small devices.

Read "Laying Out Forms" (page 55) in "Designing Forms" (page 55) for more metrics when the keyboard is displayed for user input.

Note: Although it is helpful to know the metrics on small devices like iPhone and iPod touch, you should avoid using these values in your code. Read "Using the Viewport Meta Tag" (page 42) for how to use the viewport meta tag constants.

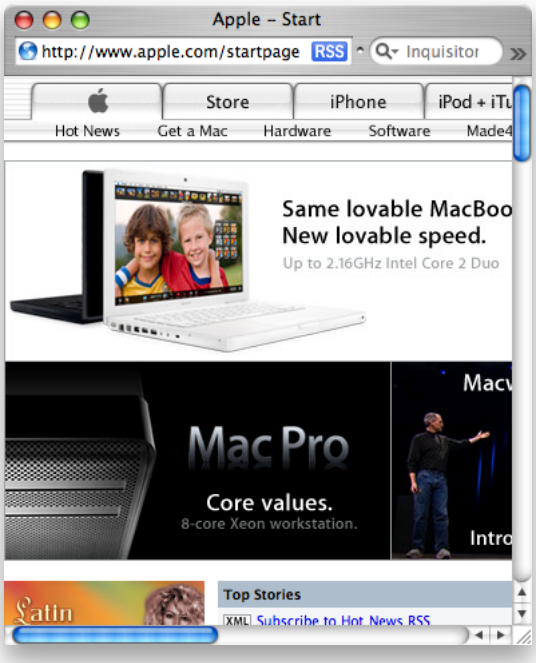
What Is the Viewport?

The viewport on the desktop and the viewport on iOS are slightly different.

Safari on iOS has no windows, scroll bars, or resize buttons as shown on the right in Figure 3-2. The user pans by flicking a finger. The user zooms in by double-tapping and pinch opening, and zooms out by pinch closing—gestures that are not available for Safari on the desktop. Because of the differences in the way users interact with web content, the viewport on the desktop and on iOS are not the same. Note that these differences between the viewports may affect some of the HTML and CSS instructions on iOS.

Figure 3-2 Differences between Safari on iOS and Safari on the desktop

Safari on the desktop

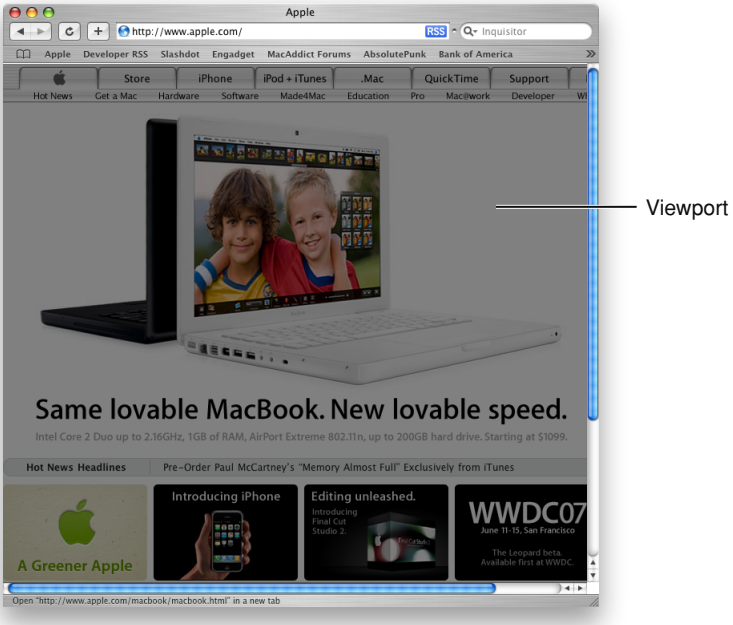


Safari on iPhone

Safari on the Desktop Viewport

The viewport on the desktop is the visible area of the webpage as shown in Figure 3-3. The user resizes the viewport by resizing the window. If the webpage is larger than the viewport, then the user scrolls to see more of the webpage. When the viewport is resized, Safari may change the document's layout—for example, expand or shrink the width of the text to fit. If the webpage is smaller than the viewport, it is filled with white space to fit the size of the viewport.

Figure 3-3 Safari on desktop viewport



Safari on iOS Viewport

For Safari on iOS, the viewport is the area that determines how content is laid out and where text wraps on the webpage. The viewport can be larger or smaller than the visible area.

When the user pans a webpage on iOS, gray bars appear on the right and bottom sides of the screen as visual feedback to show the user the size of the visible area as compared to the viewport (similar to the length of scroll bars on the desktop). Using the double tap, pinch open, and pinch close gestures, users can change the scale of the viewport but not the size. The only exception is when the user changes from portrait to landscape orientation—under certain circumstances, Safari on iOS may adjust the viewport width and height, and consequently, change the webpage layout.

You can set the viewport size and other properties of your webpage. Mostly, you do this to improve the presentation the first time iOS renders the webpage.

Examples of Viewports on iOS

The viewport on iOS is best illustrated using a few examples. Figure 3-4 shows a webpage on iPhone, containing a single 320 x 356 pixel image, that is rendered for the first time using the default viewport settings.

Figure 3-4 Viewport with default settings

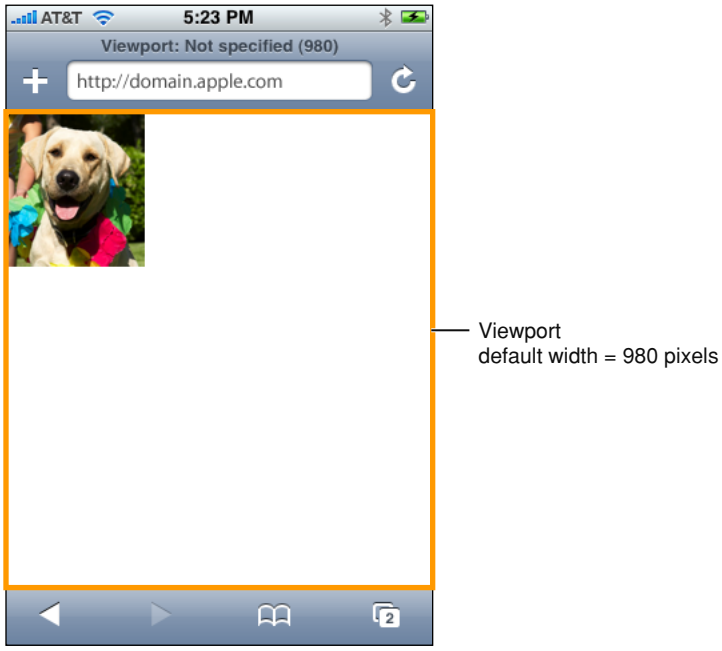
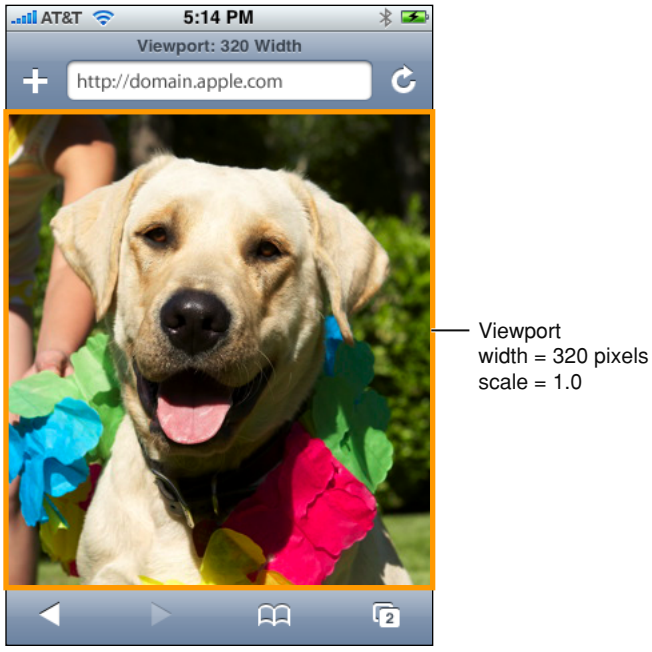


Figure 3-5 shows the same webpage with the viewport set to the size of the visible area, which is also the size of the image.

Figure 3-5 Viewport with width set to 320

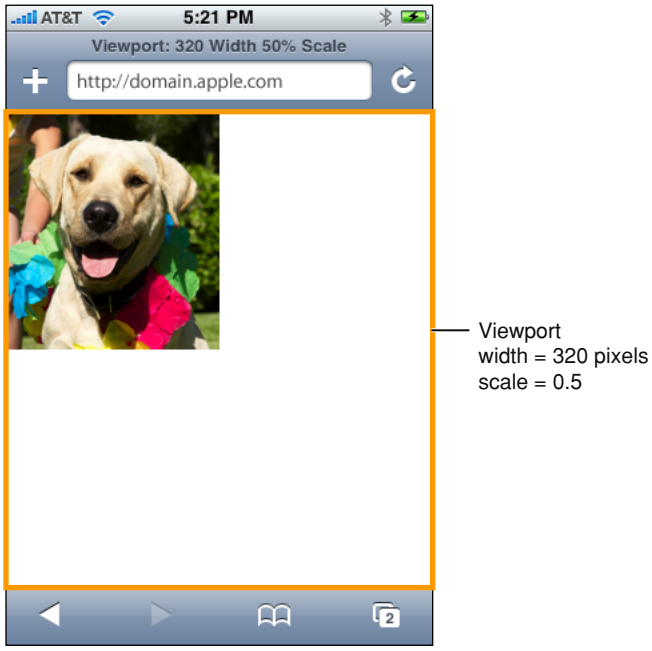


However, the viewport can be larger or smaller than the visible area. If the viewport is larger than the visible area, as shown in Figure 3-6, then the user pans to see more of the webpage.

Figure 3-6 Viewport with width set to 320 and scale set to 150%

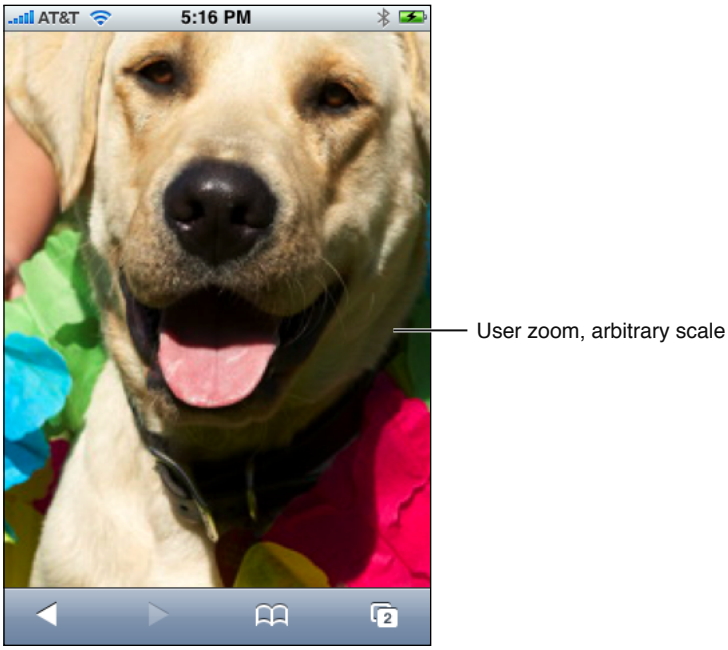
Figure 3-7 show the webpage when it is smaller than the viewport and filled with white space.

Figure 3-7 Viewport with width set to 320 and scale set to 50%



The user can also zoom in and out using gestures. When zooming in and out, the user changes the scale of the viewport, not the size of the viewport. Consequently, panning and zooming do not change the layout of the webpage. Figure 3-8 shows the same webpage when the user zooms in to see details.

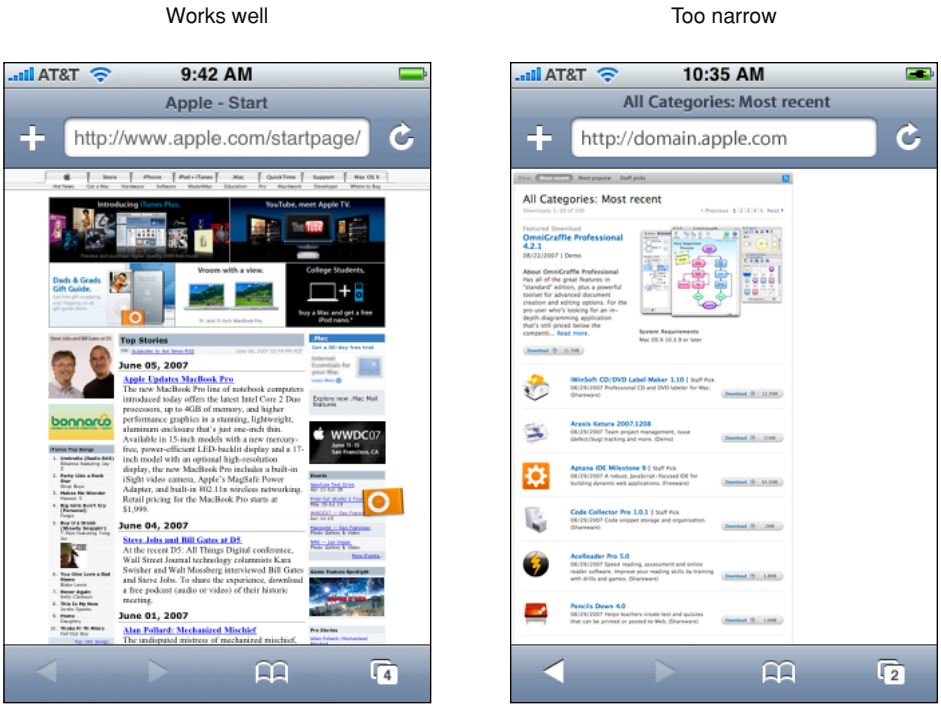
Figure 3-8 Viewport with arbitrary user scale



Default Viewport Settings

Safari on iOS sets the size and scale of the viewport to reasonable defaults that work well for most webpages, as shown on the left in Figure 3-9. The default width is 980 pixels. However, these defaults may not work well for your webpages, particularly if you are tailoring your website for a particular device. For example, the webpage on the right in Figure 3-9 appears too narrow. Because Safari on iOS provides a viewport, you can change the default settings.

Figure 3-9 Default settings work well for most webpages



Using the Viewport Meta Tag

Use the `viewport` meta tag to improve the presentation of your web content on iOS. Typically, you use the `viewport` meta tag to set the width and initial scale of the viewport. For example, if your webpage is narrower than 980 pixels, then you should set the width of the viewport to fit your web content. If you are designing an iPhone or iPod touch-specific web application, then set the width to the width of the device. Refer to "Additional meta Tag Keys" in *Safari HTML Reference* for a detailed description of the `viewport` meta tag.

Because iOS runs on devices with different screen resolutions, you should use the constants instead of numeric values when referring to the dimensions of a device. Use `device-width` for the width of the device and `device-height` for the height in portrait orientation.

You do not need to set every viewport property. If only a subset of the properties are set, then Safari on iOS infers the other values. For example, if you set the scale to `1.0`, Safari assumes the width is `device-width` in portrait and `device-height` in landscape orientation. Therefore, if you want the width to be 980 pixels and the initial scale to be 1.0, then set both of these properties.

For example, to set the viewport width to the width of the device, add this to your HTML file:

```
<meta name = "viewport" content = "width = device-width">
```

To set the initial scale to `1.0`, add this to your HTML file:

```
<meta name = "viewport" content = "initial-scale = 1.0">
```

To set the initial scale and to turn off user scaling, add this to your HTML file:

```
<meta name = "viewport" content = "initial-scale = 2.3, user-scalable = no">
```

Use the Safari on iOS console to help debug your webpages as described in "Debugging" (page 89). The console contains tips to help you choose viewport values—for example, it reminds you to use the constants when referring to the device width and height.

Changing the Viewport Width and Height

Typically, you set the viewport width to match your web content. This is the single most important optimization that you can do for iOS—make sure your webpage looks good the first time it is displayed on iOS.

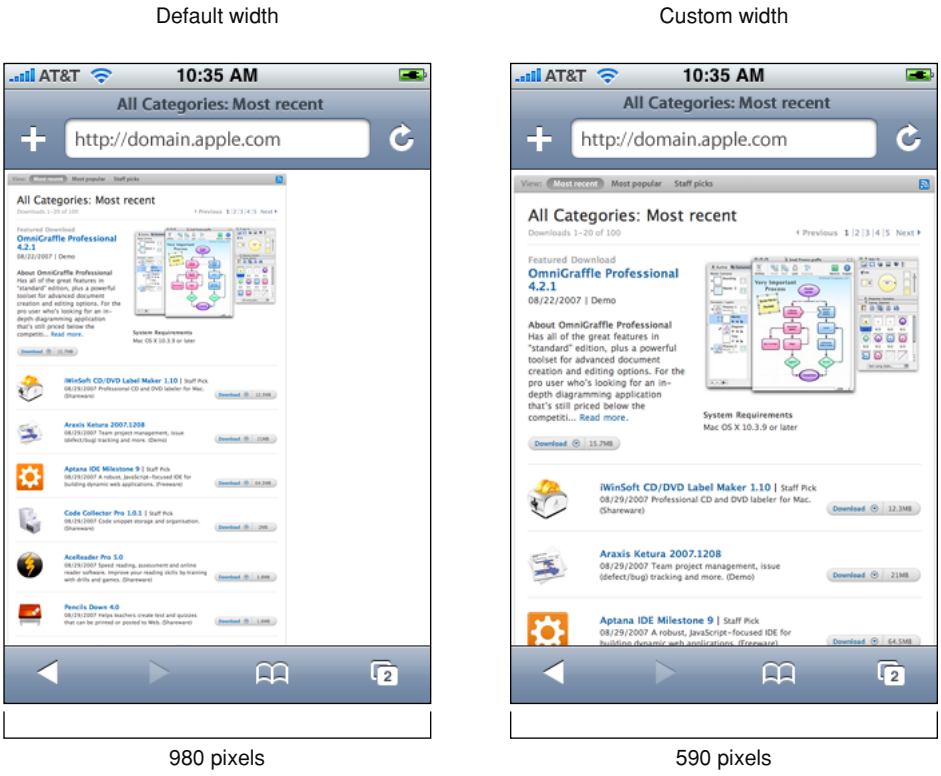
The majority of webpages fit nicely in the visible area with the viewport width set to `980` pixels in portrait orientation, as shown in Figure 3-10. If Safari on iOS did not set the viewport width to `980` pixels, then only the upper-left corner of the webpage, shown in gray, would be displayed. However, this default doesn't work for all webpages, so you'll want to use the `viewport` meta tag if your webpage is different.

Figure 3-10 Comparison of 320 and 980 viewport widths

If your webpage is narrower than the default width, as shown on the left in Figure 3-11, then set the viewport width to the width of your webpage, as shown on the right in Figure 3-11. To do this, add the following to your HTML file inside the `<head>` block, replacing `590` with the width of your webpage:

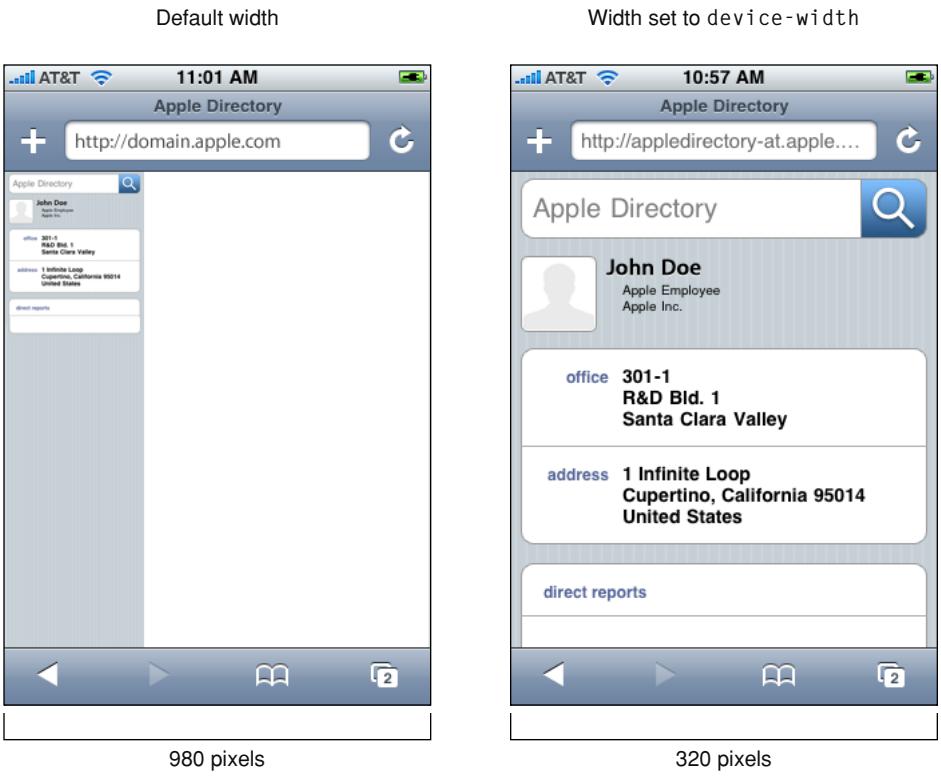
```
<meta name = "viewport" content = "width = 590">
```

Figure 3-11 Webpage is too narrow for default settings



It is particularly important to change the viewport width for web applications designed for devices with smaller screens such as iPhone and iPod touch. Figure 3-12 shows the effect of setting the viewport width to `device-width`. Read "Viewport Settings for Web Applications" (page 48) for more web application tips.

Figure 3-12 Web application page is too small for default settings



Similarly you can set the viewport height to match your web content.

How Safari Infers the Width, Height, and Initial Scale

If you set only some of the properties, then Safari on iOS infers the values of the other properties with the goal of fitting the webpage in the visible area. For example, if just the initial scale is set, then the width and height are inferred. Similarly, if just the width is set, then the height and initial scale are inferred, and so on. If the inferred values do not work for your webpage, then set more viewport properties.

Since any of the width, height, and initial scale may be inferred by Safari on iOS, the viewport may resize when the user changes orientation. For example, when the user changes from portrait to landscape orientation by rotating the device, the viewport width may expand. This is the only situation where a user action might resize the viewport, changing the layout on iOS.

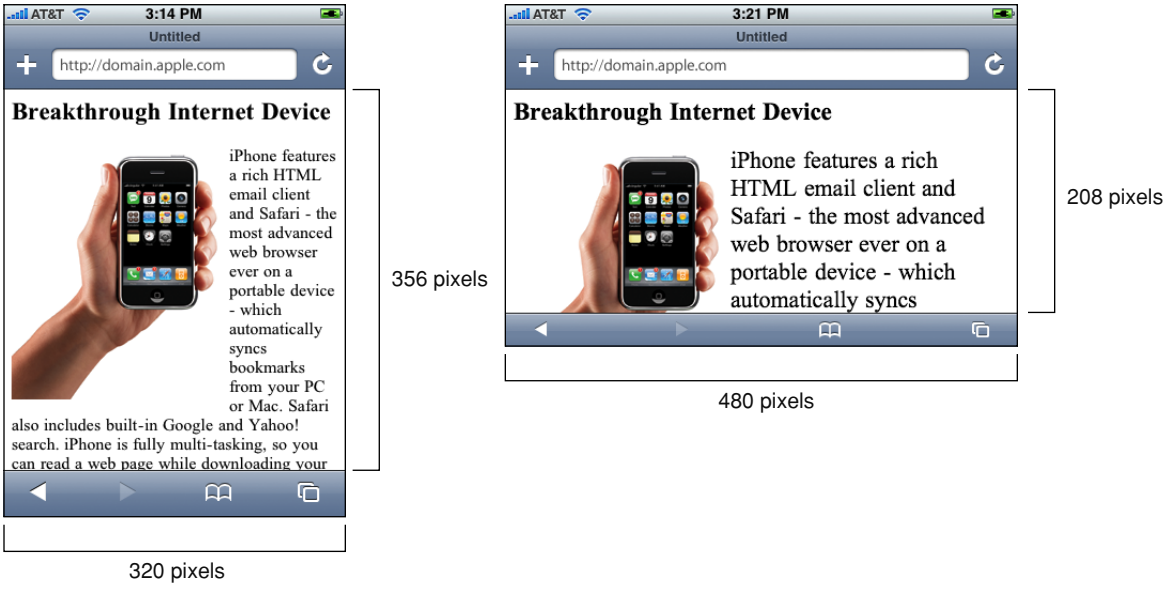
Specifically, the goal of Safari on iOS is to fit the webpage in the visible area when completely zoomed out by maintaining a ratio equivalent to the ratio of the visible area in either orientation. This is best illustrated by setting the viewport properties independently, and observing the effect on the other viewport properties. The following series of examples shows the same web content with different viewport settings.

Figure 3-13 shows a typical webpage displayed with the default settings where the viewport width is 980 and no initial scale is set.

Figure 3-13 Default width and initial scale

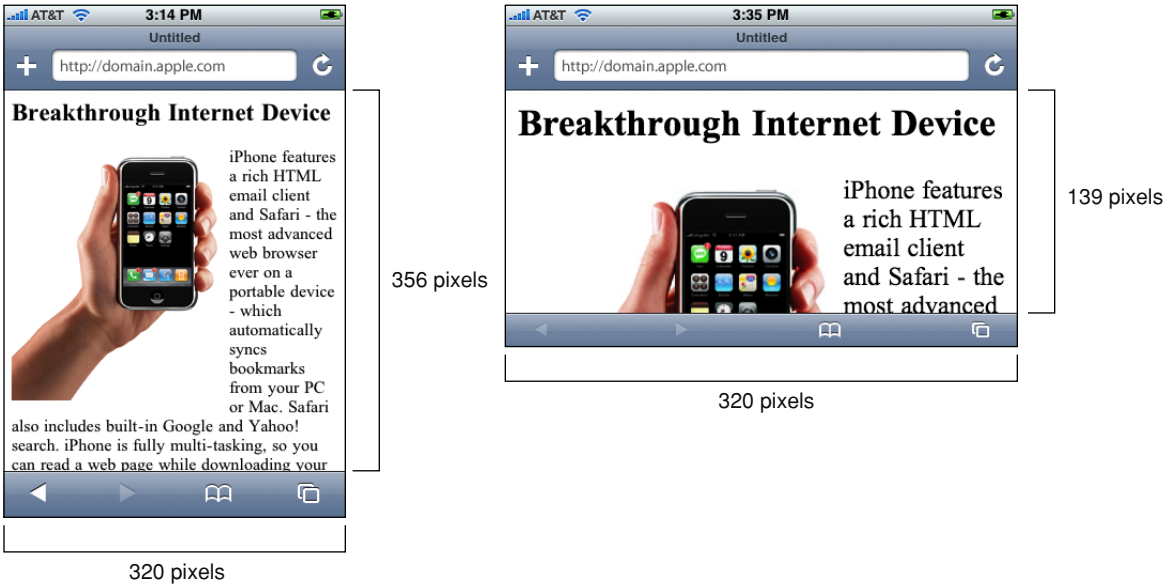
Figure 3-14 shows the same webpage when the initial scale is set to 1.0 on iPhone. Safari on iOS infers the width and height to fit the webpage in the visible area. The viewport width is set to `device-width` in portrait orientation and `device-height` in landscape orientation.

Figure 3-14 Default width with initial scale set to 1.0



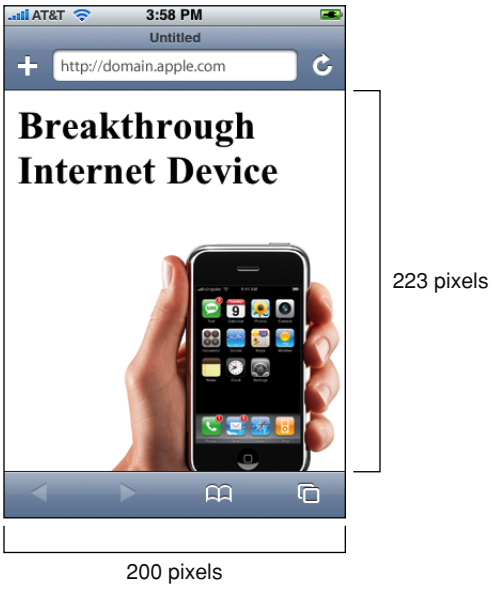
Similarly, if you specify only the viewport width, the height and initial scale are inferred. Figure 3-15 shows the rendering of the same webpage when the viewport width is set to 320 on iPhone. Notice that the portrait orientation is rendered in the same way as in Figure 3-14 (page 46), but the landscape orientation maintains a width equal to `device-width`, which changes the initial scale and has the effect of zooming in when the user changes to landscape orientation.

Figure 3-15 Width set to 320 with default initial scale



You can also set the viewport width to be smaller than the visible area with a minimum value of 200 pixels. Figure 3-16 shows the same webpage when the viewport width is set to 200 pixels on iPhone. Safari on iOS infers the height and initial scale, which has the effect of zooming in when the webpage is first rendered.

Figure 3-16 Width set to 200 with default initial scale



Finally, Figure 3-17 shows the same webpage when both the width and initial scale are set on iPhone. Safari on iOS infers the height by maintaining a ratio equivalent to the ratio of the visible area in either orientation. Therefore, if the width is set to 980 and the initial scale is set to 1.0 on iPhone, the height is set to 1091 in portrait and 425 in landscape orientation.

Figure 3-17 Width set to 980 and initial scale set to 1.0

Width = 980 pixels
Height = 1091 pixels
Initial scale = 1.0

Width = 980 pixels
Height = 425 pixels
Initial scale = 1.0

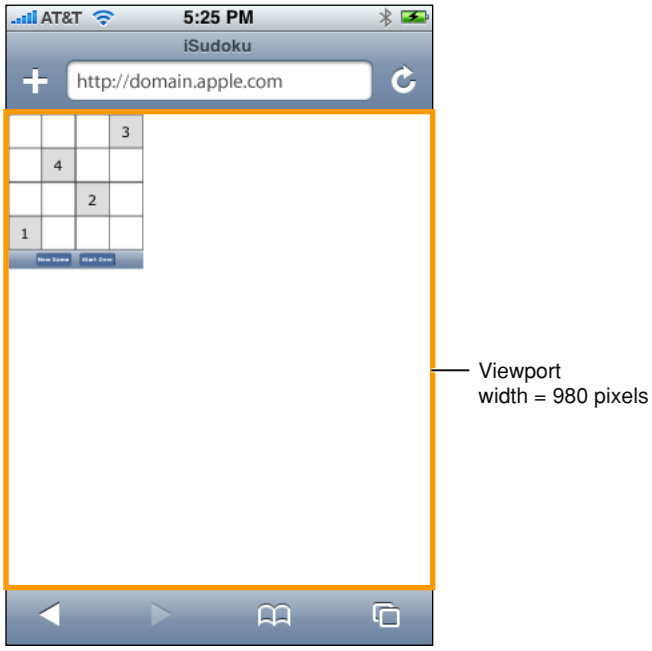
The `minimum-scale` and `maximum-scale` properties also affect the behavior when changing orientations. The range of these property values is from >0 to 10.0. The default value for both these properties is `0.25`.

Viewport Settings for Web Applications

If you are designing a web application specifically for iOS, then the recommended size for your webpages is the size of the visible area on iOS. Apple recommends that you set the width to `device-width` so that the scale is 1.0 in portrait orientation and the viewport is not resized when the user changes to landscape orientation.

If you do not change the viewport properties, Safari on iOS displays your webpage in the upper-left corner as shown in Figure 3-18. Setting the viewport width should be the first task when designing web applications for iOS to avoid the user zooming in before using your application.

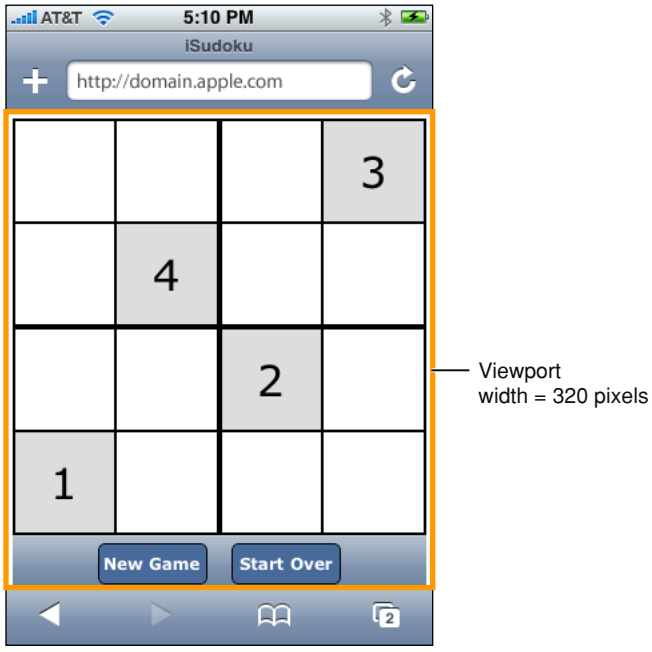
Figure 3-18 Not specifying viewport properties



By setting the width to `device-width` in portrait orientation, Safari on iOS displays your webpage as show in Figure 3-19. Users can pan down to view the rest of the webpage if it is taller than the visible area. Add this line to your HTML file to set the viewport width to `device-width`:

```
<meta name = "viewport" content = "width=device-width">
```

Figure 3-19 Width set to device-width pixels



You may not want users to scale web applications designed specifically for iOS. In this case, set the width and turn off user scaling as follows:

```
<meta name = "viewport" content = "user-scalable=no, width=device-width">
```

Customizing Style Sheets

Although configuring the viewport is an important way to optimize your web content for iOS, style sheets provide further techniques for optimizing. For example, use iOS CSS extensions to control text resizing and element highlighting. If you use conditional CSS, then you can use these settings without affecting the way other browsers render your webpages.

Read "Optimizing Web Content" (page 29) for how to use conditional CSS and "CSS Basics" (page 101) for how to add CSS to existing HTML. See *Safari CSS Reference* for a complete list of CSS properties supported by Safari.

Leveraging CSS3 Properties

There are many CSS3 properties available for you to use in Safari on the desktop and iOS. CSS properties that begin with `-webkit-` are usually proposed CSS3 properties or Apple extensions to CSS. For example, you can use the following CSS properties to emulate the iOS user interface:

`-webkit-border-image`
: Allows you to use an image as the border for a box. See "-webkit-border-image" in *Safari CSS Reference* for details.

`-webkit-border-radius`
: Creates elements with rounded corners. See "Customizing Form Controls" (page 57) for code samples. See "-webkit-border-radius" in *Safari CSS Reference* for details.

Adjusting the Text Size

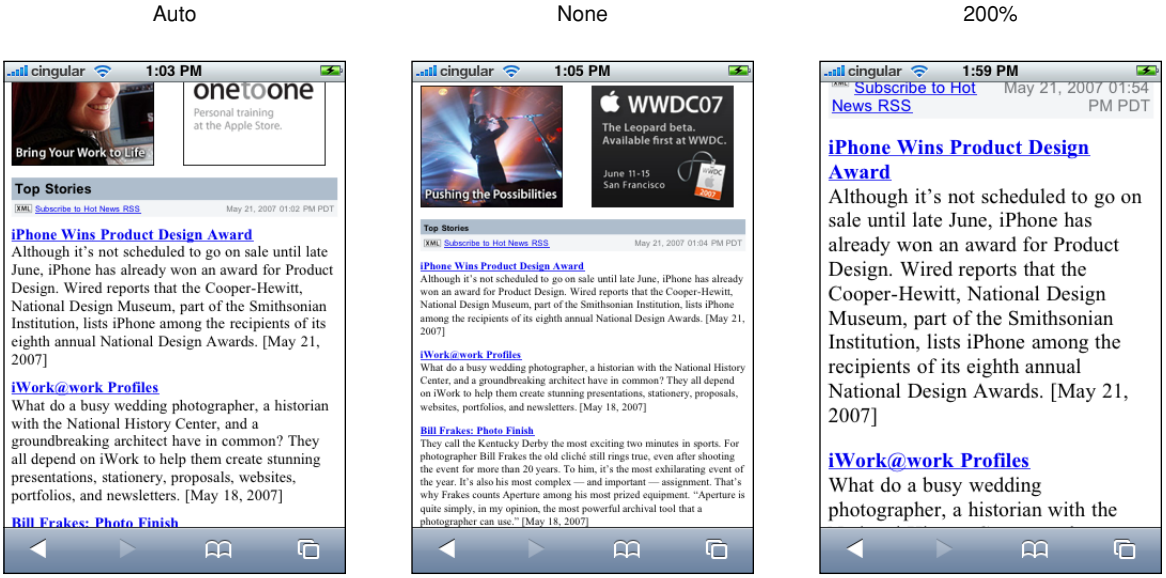
In addition to controlling the viewport, you can control the text size that Safari on iOS uses when rendering a block of text.

Adjusting the text size is important so that the text is legible when the user double-taps. If the user double-taps an HTML block element—such as a `<div>` element—then Safari on iOS scales the viewport to fit the block width in the visible area. The first time a webpage is rendered, Safari on iOS gets the width of the block and determines an appropriate text scale so that the text is legible.

If the automatic text size-adjustment doesn't work for your webpage, then you can either turn this feature off or specify your own scale as a percentage. For example, text in absolute-positioned elements might overflow the viewport after adjustment. Other pages might need a few minor adjustments to make them look better. In these cases, use the `-webkit-text-size-adjust` CSS property to change the default settings for any element that renders text.

Figure 4-1 compares a webpage rendered by Safari on iOS with `-webkit-text-size-adjust` set to `auto`, `none`, and `200%`. On iPad, the default value for `-webkit-text-size-adjust` is `none`. On all other devices, the default value is `auto`.

Figure 4-1 Comparison of text adjustment settings



To turn automatic text adjustment off, set `-webkit-text-size-adjust` to `none` as follows:

```
html {-webkit-text-size-adjust:none}
```

To change the text adjustment, set `-webkit-text-size-adjust` to a percentage value as follows, replacing `200%` with your percentage:

```
html {-webkit-text-size-adjust:200%}
```

Listing 4-1 shows setting this property for different types of blocks using inline style in HTML.

Listing 4-1 Setting the text size adjustment property

```
<body style="-webkit-text-size-adjust:none">
<table style="-webkit-text-size-adjust:auto">
<div style="-webkit-text-size-adjust:200%">
```

Highlighting Elements

By default, when the user taps a link or a JavaScript clickable element, Safari on iOS highlights the area in a transparent gray color. Using the `-webkit-tap-highlight-color` CSS property, you can either modify or disable this default behavior on your webpages.

The syntax for setting this CSS property is:

```
-webkit-tap-highlight-color:<css-color>
```

This is an inherited property that changes the tap highlight color, obeying the alpha value. If you don't specify an alpha value, Safari on iOS applies a default alpha value to the color. To disable tap highlighting, set the alpha to `0` (invisible). If you set the alpha to `1.0` (opaque), then the element won't be visible when tapped.

Listing 4-2 uses an alpha value of `0.4` for the custom highlight color shown on the right in Figure 4-2.

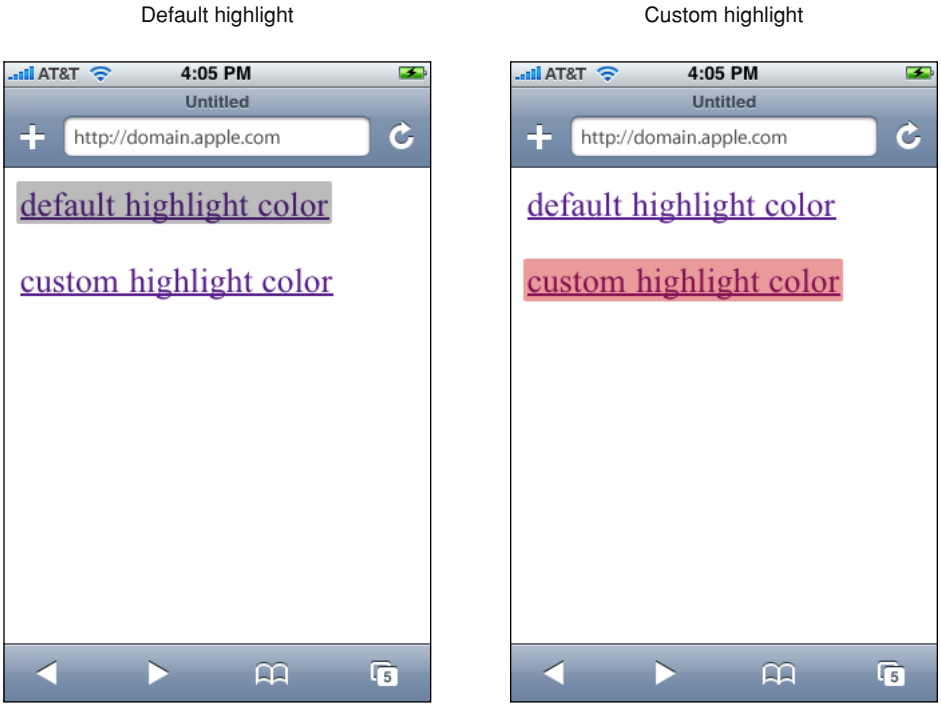
Listing 4-2 Changing the tap highlight color

```
<html>
<head>
    <meta name = "viewport" content = "width=200">
</head>

<body>
<a href = "whatever0.html">default highlight color</a><br><br>
<a href = "whatever0.html" style =
"-webkit-tap-highlight-color:rgba(200,0,0,0.4);">custom highlight color</a>
</body>

</html>
```

Figure 4-2 Differences between default and custom highlighting



Note that changing this behavior does not affect the color of the information bubble when the user touches and holds.

You can also use the `-webkit-tap-highlight-color` CSS property in combination with setting a touch event to configure buttons to behave similar to the desktop. On iOS, mouse events are sent so quickly that the down or active state is never received. Therefore, the `:active` pseudo state is triggered only when there is a touch event set on the HTML element—for example, when `ontouchstart` is set on the element as follows:

```
<button class="action" ontouchstart="" style="-webkit-tap-highlight-color:
rgba(0,0,0,0);">Testing Touch on iOS</button>
```

Now when the button is tapped and held on iOS, the button changes to the specified color without the surrounding transparent gray color appearing.

Read "Handling Events" (page 61) for the definition of a clickable element. See "-webkit-tap-highlight-color" in *Safari CSS Reference* to learn more about this property. Read "Handling Multi-Touch Events" (page 66) for details on touch events.

Designing Forms

There are many adjustments you can make to your forms so that they work better on iOS. The forms should fit neatly on the iOS screen, especially if you are designing a web application specifically for iOS. Web applications can have a rich user interface and even look like native applications to the user. Consequently, the user may expect them to behave like native applications, too.

This chapter explains what you can do to make your forms work well on iOS:

- Take into account the available screen space when the keyboard is and isn't displayed.
- Use CSS extensions to create custom controls.
- Control where automatic correction and capitalization are used.

See *iOS Human Interface Guidelines* for more tips on laying out forms and designing web applications for iOS. Read "Hiding Safari User Interface Components" (page 74) for how to use the full-screen like a native application.

Laying Out Forms

The available area for your forms changes depending on whether or not the keyboard is displayed on iOS. You should compute this area and design your forms accordingly.

Figure 5-1 shows the layout of Safari controls when the keyboard is displayed on iPhone. The status bar that appears at the top of the screen contains the time and Wi-Fi indicator. The URL text field is displayed below the status bar. The keyboard is used to enter text in forms and is displayed at the bottom of the screen. The form assistant appears above the keyboard when editing forms. It contains the Previous, Next, and Done buttons. The user taps the Next and Previous buttons to move between form elements. The user taps Done to dismiss the keyboard. The button bar contains the back, forward, bookmarks, and page buttons and appears at the bottom of the screen. The tool bar is not visible when the keyboard is visible. Your webpage is displayed in the area below the URL text field and above the tool bar or keyboard.

Figure 5-1 Form metrics when the keyboard is displayed

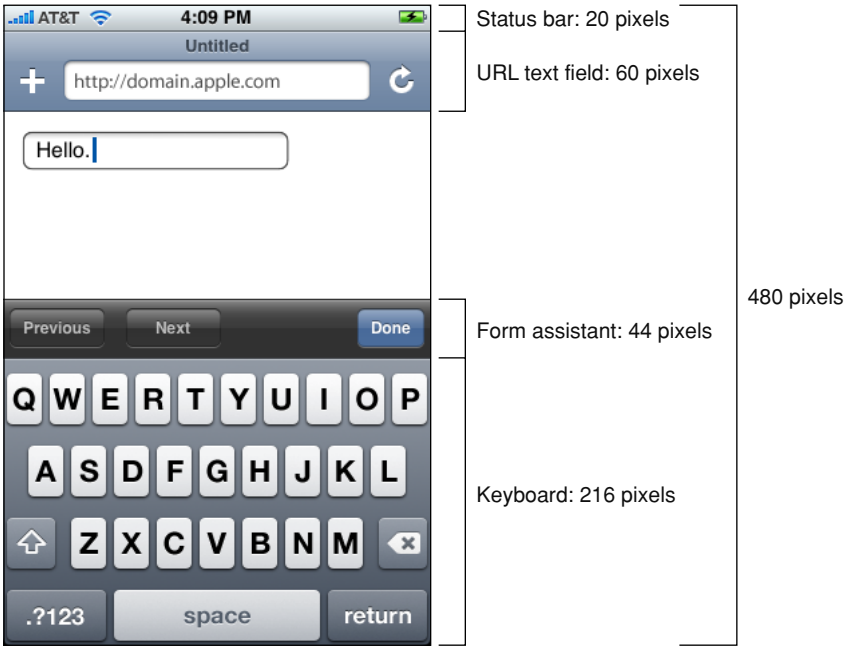


Table 5-1 contains the metrics for the objects that you need to be aware of, in both portrait and landscape orientation, when laying out forms to fit on iPhone and iPod touch.

Table 5-1 Form metrics

Object	Metrics in pixels
Status bar	Height = 20
URL text field	Height = 60
Form assistant	Height = 44
Keyboard	Portrait height = 216 Landscape height = 162
Button bar	Portrait height = 44 Landscape height = 32

Use this information to compute the available area for your web content when the keyboard is and isn't displayed. For example, when the keyboard is not displayed, the height available for your web content on iPhone is 480 - 20 - 60 - 44 = 356. Therefore, you should design your content to fit within 320 x 356 pixels in portrait orientation. If the keyboard is displayed, the available area is 320 x 140 pixels on iPhone.

iOS Note: In iOS 1.1.4 and earlier, the keyboard height in landscape orientation on iPhone and iPod touch was 180 pixels.

Customizing Form Controls

Form controls in Safari on iOS are resolution independent and can be styled with CSS specifically for iOS. You can create custom checkboxes, text fields, and select elements.

For example, you can create a custom checkbox designed for iOS as shown in Figure 5-2 with the CSS code fragment in Listing 5-1. This example uses the `-webkit-border-radius` property—an Apple extension to WebKit. See *Safari CSS Reference* for details on more WebKit properties.

Figure 5-2 A custom checkbox

Listing 5-1 Creating a custom checkbox with CSS

```
{
   width: 100px;
   height: 100px;
   -webkit-border-radius:50px;
   background-color:purple;
}
```

Figure 5-3 shows a custom text field with rounded corners corresponding to the CSS code in Listing 5-2 (page 58).

Figure 5-3 A custom text field

Listing 5-2 Creating a custom text field with CSS

```
{
    -webkit-border-radius:10px;
}
```

Figure 5-4 shows a custom select control corresponding to the CSS code in Listing 5-3 (page 59).

Figure 5-4 A custom select element

Listing 5-3 Creating a custom select control with CSS

```
{
   background:red;
   border: 1px dashed purple;
   -webkit-border-radius:10px;
}
```

Configuring Automatic Correction and Capitalization

You can also control whether or not automatic correction or capitalization are used in your forms on iOS. Set the `autocorrect` attribute to `on` if you want automatic correction and the `autocapitalize` attribute to `on` if you want automatic capitalization. If you do not set these attributes, then the browser chooses whether or not to use automatic correction or capitalization. For example, Safari on iOS turns the `autocorrect` and `autocapitalize` attributes off in login fields and on in normal text fields.

For example, the following line turn the `autocorrect` attribute on:

```
<input type="text" name="field1" autocorrect = "on"/>
```

The following line turns the `autocorrect` attribute off:

```
<input type="text" name="field2" autocorrect = "off"/>
```

You can also use this attribute on `<form>` elements to give inner form controls (like `<input>` and `<textarea>` elements) default behavior. If the inner form controls have these attributes set, those values are used instead. If they don't have these attributes set, the settings of their parent `<form>` element are used. If neither element has these attributes set, the default value is used. The default value for form elements is `on`.

For example, the following code fragment sets the `autocapitalize` attribute to `off` on the form but `on` in one of the two enclosing input elements.

```
<form action="someaction.html" method="post" autocapitalize="off">
    <!-- will have autocapitalize off, even though the default is on -->
    <input type="text" name="first-name">
    <!-- will have autocapitalize on, overriding the value in the form -->
    <input type="text" name="last-name" autocapitalize="on">
</form>
```

Handling Events

This chapter describes the events that occur when the user interacts with a webpage on iOS. Forms and documents generate the typical events in iOS that you might expect on the desktop. Gestures handled by Safari on iOS emulate mouse events. In addition, you can register for iOS-specific multi-touch and gesture events directly. Orientation events are another example of an iOS-specific event. Also, be aware that there are some unsupported events such as cut, copy, and paste.

Gestures that the user makes—for example, a double tap to zoom and a flick to pan—emulate mouse events. However, the flow of events generated by one-finger and two-finger gestures are conditional depending on whether or not the selected element is clickable or scrollable as described in "One-Finger Events" (page 61) and "Two-Finger Events" (page 64).

A clickable element is a link, form element, image map area, or any other element with `mousemove`, `mousedown`, `mouseup`, or `onclick` handlers. A scrollable element is any element with appropriate overflow style, text areas, and scrollable `iframe` elements. Because of these differences, you might need to change some of your elements to clickable elements, as described in "Making Elements Clickable" (page 65), to get the desired behavior in iOS.

In addition, you can turn off the default Safari on iOS behavior as described in "Preventing Default Behavior" (page 69) and handle your own multi-touch and gesture events directly. Handling multi-touch and gesture events directly gives developers the ability to implement unique touch-screen interfaces similar to native applications. Read "Handling Multi-Touch Events" (page 66) and "Handling Gesture Events" (page 68) to learn more about DOM touch events.

If you want to change the layout of your webpage depending on the orientation of iOS, read "Handling Orientation Events" (page 69).

See "Supported Events" (page 71) for a complete list of events supported in iOS.

On iOS, emulated mouse events are sent so quickly that the down or active pseudo state of buttons may never occur. Read "Highlighting Elements" (page 52) for how to customize a button to behave similar to the desktop.

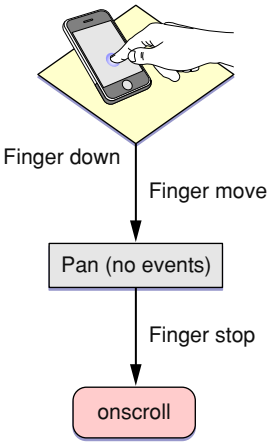
It's very common to combine DOM touch events with CSS visual effects. Read *Safari CSS Visual Effects Guide* to learn more about CSS visual effects.

One-Finger Events

This section uses flow charts to break down gestures into the individual actions that might generate events. Some of the events generated on iOS are conditional—the events generated depend on what the user is tapping or touching and whether they are using one or two fingers. Some gestures don't generate any events on iOS.

One-finger panning doesn't generate any events until the user stops panning—an `onscroll` event is generated when the page stops moving and redraws—as shown in Figure 6-1.

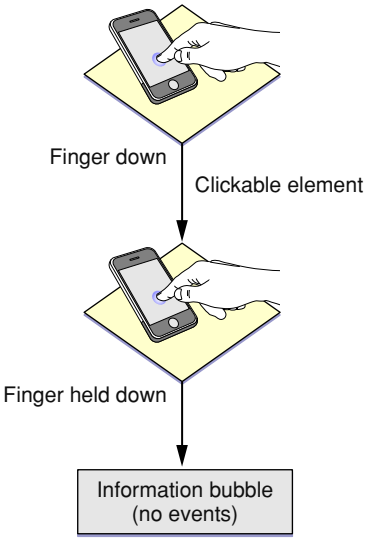
Figure 6-1 The panning gesture



Displaying the information bubble doesn't generate any events as shown in Figure 6-2. However, if the user touches and holds an image, the image save sheet appears instead of an information bubble.

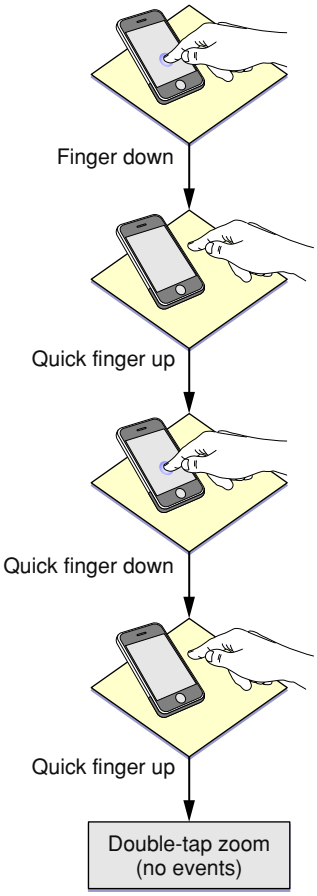
iOS Note: The image save sheet appears on iOS 2.0 and later.

Figure 6-2 The touch and hold gesture



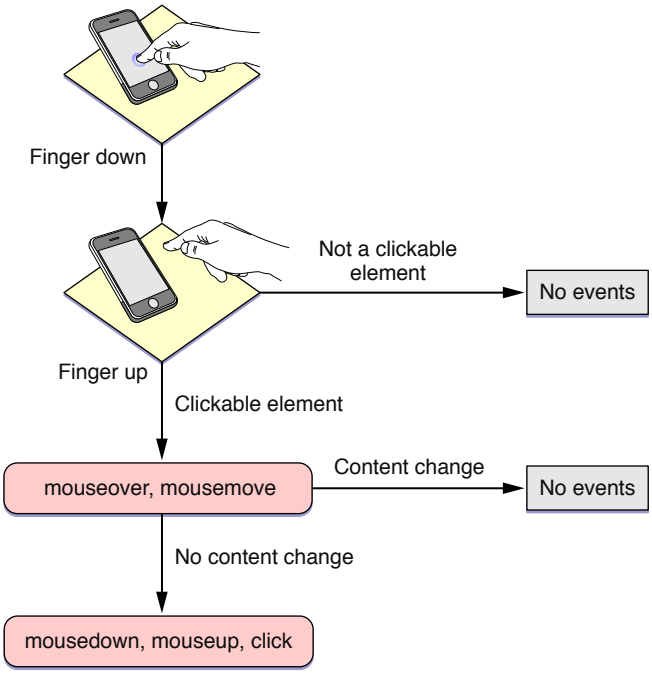
Finally, a double tap doesn't generate any events either as shown in Figure 6-3.

Figure 6-3 The double-tap gesture



Mouse events are delivered in the same order you'd expect in other web browsers illustrated in Figure 6-4. If the user taps a nonclickable element, no events are generated. If the user taps a clickable element, events arrive in this order: `mouseover`, `mousemove`, `mousedown`, `mouseup`, and `click`. The `mouseout` event occurs only if the user taps on another clickable item. Also, if the contents of the page changes on the `mousemove` event, no subsequent events in the sequence are sent. This behavior allows the user to tap in the new content.

Figure 6-4 One-finger gesture emulating a mouse



Two-Finger Events

The pinch open gesture does not generate any mouse events as shown in Figure 6-5.

Figure 6-5 The pinch open gesture

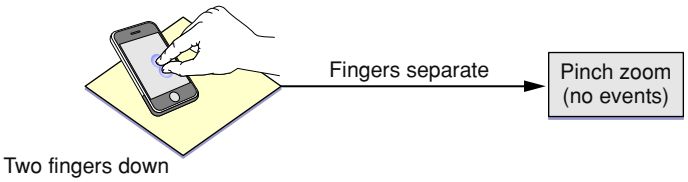
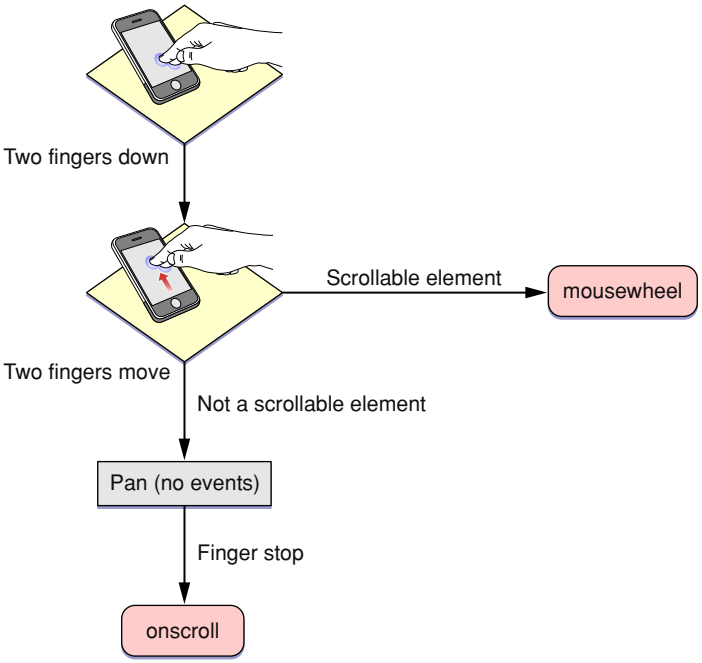


Figure 6-6 illustrates the mouse events generated by using two fingers to pan a scrollable element. The flow of events is as follows:

- If the user holds two fingers down on a scrollable element and moves the fingers, `mousewheel` events are generated.
- If the element is not scrollable, Safari on iOS pans the webpage. No events are generated while panning.
- An `onscroll` event is generated when the user stops panning.

Figure 6-6 Two-finger panning gesture



Form and Document Events

Typical events generated by forms and documents include `blur`, `focus`, `load`, `unload`, `reset`, `submit`, `change` and `abort`. See "Supported Events" (page 71) for a complete list of supported events on iOS.

Making Elements Clickable

Because of the way Safari on iOS creates events to emulate a mouse, some of your elements may not behave as expected on iOS. In particular, some menus that only use `mousemove` handlers, as in Listing 6-1, need to be changed because iOS doesn't recognize them as clickable elements.

Listing 6-1 A menu using a mouseover handler

```
<span onmouseover = "..."
  onmouseout  = "..."

WHERE TO BUY

</span>
```

To fix this, add a dummy `onclick` handler, `onclick = "void(0)"`, so that Safari on iOS recognizes the `span` element as a clickable element, as shown in Listing 6-2.

Listing 6-2 Adding an onclick handler

```
<span onmouseover = "..."
  onmouseout  = "..."
  onclick = "void(0)">

WHERE TO BUY

</span>
```

Handling Multi-Touch Events

You can use JavaScript DOM touch event classes available on iOS to handle multi-touch and gesture events in a way similar to the way they are handled in native iOS applications.

If you register for multi-touch events, the system continually sends `TouchEvent` objects to those DOM elements as fingers touch and move across a surface. These are sent in addition to the emulated mouse events unless you prevent this default behavior as described in "Preventing Default Behavior" (page 69). A touch event provides a snapshot of all touches during a multi-touch sequence, most importantly the touches that are new or have changed for a particular target. The different types of multi-touch events are described in TouchEvent in *Safari DOM Additions Reference*.

A multi-touch sequence begins when a finger first touches the surface. Other fingers may subsequently touch the surface, and all fingers may move across the surface. The sequence ends when the last of these fingers is lifted from the surface. An application receives touch event objects during each phase of any touch.

Touch events are similar to mouse events except that you can have simultaneous touches on the screen at different locations. A touch event object is used to encapsulate all the touches that are currently on the screen. Each finger is represented by a touch object. The typical properties that you find in a mouse event are in the touch object, not the touch event object.

Note that a sequence of touch events is delivered to the element that received the original `touchstart` event regardless of the current location of the touches.

Follow these steps to use multi-touch events in your web application.

1. Register handlers for multi-touch events in HTML as follows:

```
<div
  ontouchstart="touchStart(event);"
  ontouchmove="touchMove(event);"
  ontouchend="touchEnd(event);"
  ontouchcancel="touchCancel(event);"
></div>
```

2. Alternatively, register handlers in JavaScript as follows:

```
element.addEventListener("touchstart", touchStart, false);
element.addEventListener("touchmove", touchMove, false);
element.addEventListener("touchend", touchEnd, false);
element.addEventListener("touchcancel", touchCancel, false);
```

3. Respond to multi-touch events by implementing handlers in JavaScript.

For example, implement the `touchStart` method as follows:

```
function touchStart(event) {
  // Insert your code here
}
```

4. Optionally, get all touches on a page using the `touches` property as follows:

```
var allTouches = event.touches;
```

Note that you can get all other touches for an event even when the event is triggered by a single touch.

5. Optionally, get all touches for the target element using the `targetTouches` property:

```
var targetTouches = event.targetTouches;
```

6. Optionally, get all changed touches for this event using the `changedTouches` property:

```
var changedTouches = event.changedTouches;
```

7. Access the `Touch` object properties—such as the target, identifier, and location in page, client, or screen coordinates—similar to mouse event properties.

For example, get the number of touches:

```
event.touches.length
```

Get a specific touch object at index `i`:

```
var touch = event.touches[i];
```

Finally, get the location in page coordinates for a single-finger event:

```
var x = event.touches[0].pageX;
var y = event.touches[0].pageY;
```

You can also combine multi-touch events with CSS visual effects to enable dragging or some other user action. To enable dragging, implement the `touchmove` event handler to translate the target:

```
function touchMove(event) {
    event.preventDefault();
    curX = event.targetTouches[0].pageX - startX;
    curY = event.targetTouches[0].pageY - startY;
    event.targetTouches[0].target.style.webkitTransform =
        'translate(' + curX + 'px, ' + curY + 'px)';
}
```

Typically, you implement multi-touch event handlers to track one or two touches. But you can also use multi-touch event handlers to identify custom gestures. That is, custom gestures that are not already identified for you by gesture events described in "Handling Gesture Events" (page 68). For example, you can identify a two-finger tap gesture as follows:

1. Begin gesture if you receive a `touchstart` event containing two target touches.

2. End gesture if you receive a `touchend` event with no preceding `touchmove` events.

Similarly, you can identify a swipe gesture as follows:

1. Begin gesture if you receive a `touchstart` event containing one target touch.
2. Abort gesture if, at any time, you receive an event with >1 touches.
3. Continue gesture if you receive a `touchmove` event mostly in the x-direction.
4. Abort gesture if you receive a `touchmove` event mostly the y-direction.
5. End gesture if you receive a `touchend` event.

Handling Gesture Events

Multi-touch events can be combined together to form high-level gesture events.

`GestureEvent` objects are also sent during a multi-touch sequence. Gesture events contain scaling and rotation information allowing gestures to be combined, if supported by the platform. If not supported, one gesture ends before another starts. Listen for `GestureEvent` objects if you want to respond to gestures only, not process the low-level `TouchEvent` objects. The different types of gesture events are described in GestureEvent in *Safari DOM Additions Reference*.

Follow these steps to use gesture events in your web application.

1. Register handlers for gesture events in HTML:

```
<div
  ongesturestart="gestureStart(event);"
  ongesturechange="gestureChange(event);"
  ongestureend="gestureEnd(event);"
></div>
```

2. Alternatively, register handlers in JavaScript:

```
element.addEventListener("gesturestart", gestureStart, false);
element.addEventListener("gesturechange", gestureChange, false);
element.addEventListener("gestureend", gestureEnd, false);
```

3. Respond to gesture events by implementing handlers in JavaScript.

   For example, implement the `gestureChange` method as follows:

```
function gestureChange(event) {
  // Insert your code here
}
```

4. Get the amount of rotation since the gesture started:

```
var angle = event.rotation;
```

   The angle is in degrees, where clockwise is positive and counterclockwise is negative.

5. Get the amount scaled since the gesture started:

```
var scale = event.scale;
```

The scale is smaller if less than `1.0` and larger if greater than `1.0`.

You can combine gesture events with CSS visual effects to enable scaling, rotating, or some other custom user action. For example, implement the `gesturechange` event handler to scale and rotate the target as follows:

```
onGestureChange: function(e) {
    e.preventDefault();
    e.target.style.webkitTransform =
        'scale(' + e.scale  + startScale  + ') rotate(' + e.rotation +
startRotation + 'deg)';
}
```

Preventing Default Behavior

iOS Note: The `preventDefault` method applies to multi-touch and gesture input in iOS 2.0 and later.

The default behavior of Safari on iOS can interfere with your application's custom multi-touch and gesture input. You can disable the default browser behavior by sending the `preventDefault` message to the event object.

For example, to prevent scrolling on an element in iOS 2.0, implement the `touchmove` and `touchstart` event handlers as follows :

```
function touchMove(event) {
    // Prevent scrolling on this element
    event.preventDefault();
    ...
}
```

To disable pinch open and pinch close gestures in iOS 2.0, implement the `gesturestart` and `gesturechange` event handlers as follows:

```
function gestureChange(event) {
    // Disable browser zoom
    event.preventDefault();
    ...
}
```

Important: The default browser behavior may change in future releases.

Handling Orientation Events

An event is sent when the user changes the orientation of iOS. By handling this event in your web content, you can determine the current orientation of the device and make layout changes accordingly. For example, display a simple textual list in portrait orientation and add a column of icons in landscape orientation.

Similar to a `resize` event, a handler can be added to the `<body>` element in HTML as follows:

```
<body onorientationchange="updateOrientation();">
```

where `updateOrientation` is a handler that you implement in JavaScript.

In addition, the `window` object has an `orientation` property set to either `0`, `-90`, `90`, or `180`. For example, if the user starts with the iPhone in portrait orientation and then changes to landscape orientation by turning the iPhone to the right, the window's `orientation` property is set to `-90`. If the user instead changes to landscape by turning the iPhone to the left, the window's `orientation` property is set to `90`.

Listing 6-3 adds an orientation handler to the body element and implements the `updateOrientation` JavaScript method to display the current orientation on the screen. Specifically, when an `orientationchange` event occurs, the `updateOrientation` method is invoked, which changes the string displayed by the division element in the body.

Listing 6-3 Displaying the orientation

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Orientation</title>
        <meta name = "viewport" content="width=320, user-scalable=0"/>

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

            function updateOrientation()
            {
                var displayStr = "Orientation : ";

                switch(window.orientation)
                {
                    case 0:
                        displayStr += "Portrait";
                    break;

                    case -90:
                        displayStr += "Landscape (right, screen turned
clockwise)";
                    break;

                    case 90:
                        displayStr += "Landscape (left, screen turned
counterclockwise)";
                    break;

                    case 180:
                        displayStr += "Portrait (upside-down portrait)";
                    break;

                }
                document.getElementById("output").innerHTML = displayStr;
            }
    </script>
    </head>
```

```
    <body onorientationchange="updateOrientation();">
        <div id="output"></div>
    </body>
</html>
```

Supported Events

Be aware of all the events that iOS supports and under what conditions they are generated. Table 6-1 specifies which events are generated by Safari on iOS and which are generated conditionally depending on the type of element selected. This table also lists unsupported events.

iOS Note: Although drag and drop are not supported, you can produce the same effect using touch events as described in "Using Touch to Drag Elements" in *Safari CSS Visual Effects Guide*.

Table 6-1 Types of events

Event	Generated	Conditional	Available
`mousemove`	Yes	Yes	iOS 1.0 and later.
`mousedown`	Yes	Yes	iOS 1.0 and later.
`mouseup`	Yes	Yes	iOS 1.0 and later.
`mouseover`	Yes	Yes	iOS 1.0 and later.
`mouseout`	Yes	Yes	iOS 1.0 and later.
`click`	Yes	Yes	iOS 1.0 and later.
`blur`	Yes	No	iOS 1.0 and later.
`focus`	Yes	No	iOS 1.0 and later.
`load`	Yes	No	iOS 1.0 and later.
`unload`	Yes	No	iOS 1.0 and later.
`reset`	Yes	No	iOS 1.0 and later.
`submit`	Yes	No	iOS 1.0 and later.
`change`	Yes	No	iOS 1.0 and later.
`abort`	Yes	No	iOS 1.0 and later.
`cut`	No	N/A	
`copy`	No	N/A	
`paste`	No	N/A	

Event	Generated	Conditional	Available
`selection`	No	N/A	
`drag`	No	N/A	
`drop`	No	N/A	
`orientationchange`	Yes	N/A	iOS 1.1.1 and later.
`touchstart`	Yes	N/A	iOS 2.0 and later.
`touchmove`	Yes	N/A	iOS 2.0 and later.
`touchend`	Yes	N/A	iOS 2.0 and later.
`touchcancel`	Yes	N/A	iOS 2.0 and later.
`gesturestart`	Yes	N/A	iOS 2.0 and later.
`gesturechange`	Yes	N/A	iOS 2.0 and later.
`gestureend`	Yes	N/A	iOS 2.0 and later.

Configuring Web Applications

A web application is designed to look and behave in a way similar to a native application—for example, it is scaled to fit the entire screen on iOS. You can tailor your web application for Safari on iOS even further, by making it appear like a native application when the user adds it to the Home screen. You do this by using settings for iOS that are ignored by other platforms.

For example, you can specify an icon for your web application used to represent it when added to the Home screen on iOS, as described in "Specifying a Webpage Icon for Web Clip" (page 73). You can also minimize the Safari on iOS user interface, as described in "Changing the Status Bar Appearance" (page 75) and "Hiding Safari User Interface Components" (page 74), when your web application is launched from the Home screen. These are all optional settings that when added to your web content are ignored by other platforms.

Read "Viewport Settings for Web Applications" (page 48) for how to set the viewport for web applications on iOS.

Specifying a Webpage Icon for Web Clip

iOS Note: The Web Clip feature is available in iOS 1.1.3 and later. The `apple-touch-icon-precomposed.png` filename is available in iOS 2.0 and later. Support for multiple icons for different device resolutions is available in iOS 4.2 and later.

You may want users to be able to add your web application or webpage link to the Home screen. These links, represented by an icon, are called Web Clips. Follow these simple steps to specify an icon to represent your web application or webpage on iOS.

- To specify an icon for the entire website (every page on the website), place an icon file in PNG format in the root document folder called `apple-touch-icon.png` or `apple-touch-icon-precomposed.png`. If you use `apple-touch-icon-precomposed.png` as the filename, Safari on iOS won't add any effects to the icon.
- To specify an icon for a single webpage or replace the website icon with a webpage-specific icon, add a link element to the webpage, as in:

  ```
  <link rel="apple-touch-icon" href="/custom_icon.png"/>
  ```

  In the above example, replace `custom_icon.png` with your icon filename. If you don't want Safari on iOS to add any effects to the icon, replace `apple-touch-icon` with `apple-touch-icon-precomposed`.
- To specify multiple icons for different device resolutions—for example, support both iPhone and iPad devices—add a `sizes` attribute to each link element as follows:

  ```
  <link rel="apple-touch-icon" href="touch-icon-iphone.png" />
  <link rel="apple-touch-icon" sizes="72x72" href="touch-icon-ipad.png" />
  <link rel="apple-touch-icon" sizes="114x114" href="touch-icon-iphone4.png" />
  ```

The icon that is the most appropriate size for the device is used. If no `sizes` attribute is set, the element's size defaults to 57 x 57.

If there is no icon that matches the recommended size for the device, the smallest icon larger than the recommended size is used. If there are no icons larger than the recommended size, the largest icon is used. If multiple icons are suitable, the icon that has the `precomposed` keyword is used.

If no icons are specified using a link element, the website root directory is searched for icons with the `apple-touch-icon...` or `apple-touch-icon-precomposed...` prefix. For example, if the appropriate icon size for the device is 57 x 57, the system searches for filenames in the following order:

1. apple-touch-icon-57x57-precomposed.png
2. apple-touch-icon-57x57.png
3. apple-touch-icon-precomposed.png
4. apple-touch-icon.png

See "Custom Icon and Image Creation Guidelines" for webpage icon metrics.

Specifying a Startup Image

iOS Note: Specifying a startup image is available in iOS 3.0 and later.

On iOS, similar to native applications, you can specify a startup image that is displayed while your web application launches. This is especially useful when your web application is offline. By default, a screenshot of the web application the last time it was launched is used. To set another startup image, add a link element to the webpage, as in:

```
<link rel="apple-touch-startup-image" href="/startup.png">
```

In the above example, replace `startup.png` with your startup screen filename. On iPhone and iPod touch, the image must be 320 x 460 pixels and in portrait orientation.

Hiding Safari User Interface Components

As part of optimizing your web application for Safari on iOS, have it launch in full-screen mode to look like a native application. When using full-screen mode, Safari is not used to display the web content—specifically, there is no browser URL text field at the top of the screen or button bar at the bottom of the screen. Only a status bar appears at the top of the screen. Read "Changing the Status Bar Appearance" (page 75) for how to minimize the status bar.

Set the `apple-mobile-web-app-capable` meta tag to `yes` to turn on this feature. For example, the following HTML displays web content in full-screen mode.

```
<meta name="apple-mobile-web-app-capable" content="yes" />
```

You can determine whether a webpage is displayed in full-screen mode using the `window.navigator.standalone` read-only Boolean JavaScript property.

Changing the Status Bar Appearance

If your web application displays in full-screen mode like that of a native application, you can minimize the status bar that is displayed at the top of the screen on iOS. Do so using the status-bar-style meta tag.

This meta tag has no effect unless you first specify full-screen mode as described in "Hiding Safari User Interface Components" (page 74). Then use the status bar style meta tag, `apple-mobile-web-app-status-bar-style`, to change the appearance of the status bar depending on your application needs. For example, if you want to use the entire screen, set the status bar style to translucent black.

For example, the following HTML sets the background color of the status bar to black:

```
<meta name="apple-mobile-web-app-status-bar-style" content="black" />
```

Creating Video

Safari supports audio and video viewing in a webpage on the desktop and iOS. You can use `audio` and `video` HTML elements or use the `embed` element to use the native application for video playback. In either case, you need to ensure that the video you create is optimized for the platform and different bandwidths.

iOS streams movies and audio using HTTP over EDGE, 3G, and Wi-Fi networks. iOS uses a native application to play back video even when video is embedded in your webpages. Video automatically expands to the size of the screen and rotates when the user changes orientation. The controls automatically hide when they are not in use and appear when the user taps the screen. This is the experience the user expects when viewing all video on iOS.

Safari on iOS supports a variety of rich media, including QuickTime movies, as described in "Use Supported iOS Rich Media MIME Types" (page 24). Safari on iOS does not support Flash so don't bring up JavaScript alerts that ask users to download Flash. Also, don't use JavaScript movie controls to play back video since iOS supplies its own controls.

Safari on the desktop supports the same audio and video formats as Safari on iOS. However, if you use the `audio` and `video` HTML elements on the desktop, you can customize the play back controls. See *Safari DOM Additions Reference* for more details on the `HTMLMediaElement` class.

Follow these guidelines to deliver the best web audio and video experience in Safari on any platform:

- Follow current best practices for embedding movies in webpages as described in "Sizing Movies Appropriately" (page 78), "Don't Let the Bit Rate Stall Your Movie" (page 78), and "Using Supported Movie Standards" (page 78).
- Use QuickTime Pro to encode H.264/AAC at appropriate sizes and bit rates for EDGE, 3G, and Wi-Fi networks, as described in "Encoding Video for Wi-Fi, 3G, and EDGE" (page 78).
- Use reference movies so that iOS automatically streams the best version of your content for the current network connection, as described in "Creating a Reference Movie" (page 79).
- Use poster JPEGs (not poster frames in a movie) to display a preview of your embedded movie in webpages, as described in "Creating a Poster Image for Movies" (page 80).
- Make sure the HTTP servers hosting your media files support byte-range requests, as described in "Configuring Your Server" (page 81).
- If your site has a custom media player, also provide direct links to the media files. iOS users can follow these links to play those files directly.

Sizing Movies Appropriately

In landscape orientation on iOS, the screen is 480 x 320 pixels. Users can easily switch the view mode between scaled-to-fit (letterboxed) and full-screen (centered and cropped). You should use a size that preserves the aspect ratio of your content and fits within a 480 x 360 rectangle. 480 x 360 is a good choice for 4:3 aspect ratio content and 480 x 270 is a good choice for widescreen content as it keeps the video sharp in full-screen view mode. You can also use 640 x 360 or anamorphic 640 x 480 with pixel aspect ratio tagging for widescreen content.

Don't Let the Bit Rate Stall Your Movie

When viewing media over the network, the bit rate makes a crucial difference to the playback experience. If the network cannot keep up with the media bit rate, playback stalls. Encode your media for iOS as described in "Encoding Video for Wi-Fi, 3G, and EDGE" (page 78) and use a reference movie as described in "Creating a Reference Movie" (page 79).

Using Supported Movie Standards

The following compression standards are supported:

- H.264 Baseline Profile Level 3.0 video, up to 640 x 480 at 30 fps. Note that B frames are not supported in the Baseline profile.
- MPEG-4 Part 2 video (Simple Profile)
- AAC-LC audio, up to 48 kHz

Movie files with the extensions `.mov`, `.mp4`, `.m4v`, and `.3gp` are supported.

Any movies or audio files that can play on iPod play correctly on iPhone.

If you export your movies using QuickTime Pro 7.2, as described in "Encoding Video for Wi-Fi, 3G, and EDGE" (page 78), then you can be sure that they are optimized to play on iOS.

Encoding Video for Wi-Fi, 3G, and EDGE

Because users may be connected to the Internet via wired or wireless technology, using either Wi-Fi, 3G, or EDGE on iOS, you need to provide alternate media for these different connection speeds. You can use QuickTime Pro, the QuickTime API, or any Apple applications that provide iOS exporters to encode your video for Wi-Fi, 3G, and EDGE. This section contains specific instructions for exporting video using QuickTime Pro.

Follow these steps to export video using QuickTime Pro 7.2.1 and later:

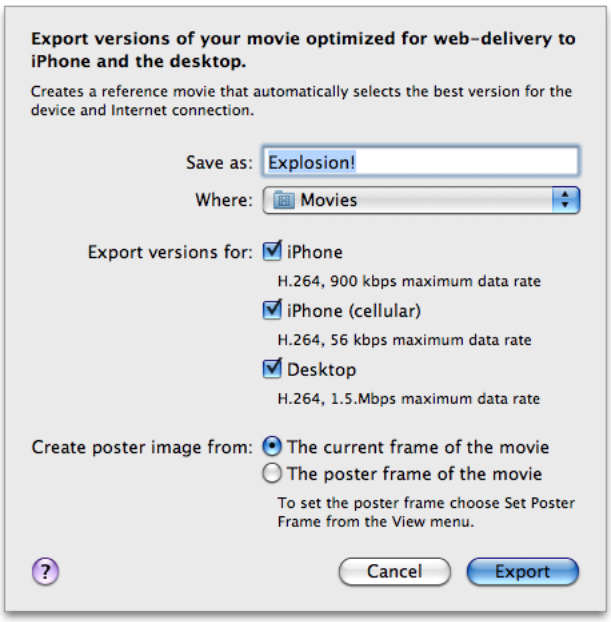
1. Open your movie using QuickTime Player Pro.

2. Choose File > Export for Web.

   A dialog appears.

3. Enter the file name prefix, location of your export, and set of versions to export as shown in Figure 8-1.

Figure 8-1 Export movie panel



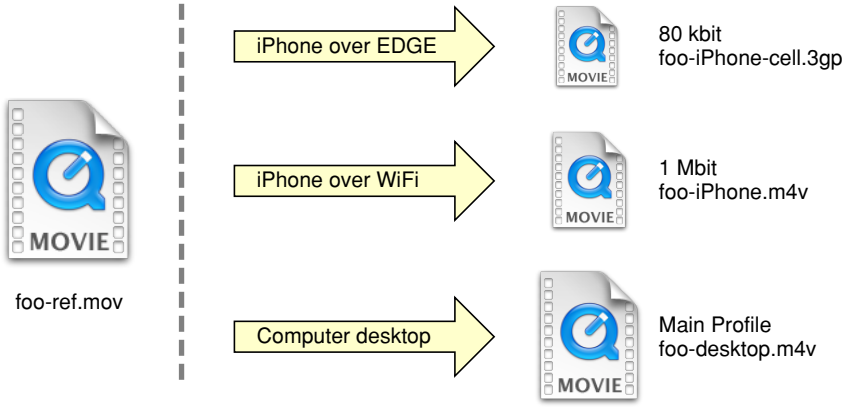
4. Click Export.

   QuickTime Player Pro saves these versions of your QuickTime movie, along with a reference movie, poster image, and `ReadMe.html` file to the specified location. See the `ReadMe.html` file for instructions on embedding the generated movie in your webpage, including sample HTML.

Creating a Reference Movie

A reference movie contains a list of movie URLs, each of which has a list of tests, as show in Figure 8-2. When opening the reference movie, a playback device or computer chooses one of the movie URLs by finding the last one that passes all its tests. Tests can check the capabilities of the device or computer and the speed of the network connection.

Figure 8-2 Reference movie components



If you use QuickTime Pro 7.2.1 or later to export your movies for iOS, as described in "Encoding Video for Wi-Fi, 3G, and EDGE" (page 78), then you already have a reference movie. Otherwise, you can use the MakeRefMovie tool to create reference movies. For more information on creating reference movies see *Creating Reference Movies - MakeRefMovie*.

Also, refer to the *MakeiPhoneRefMovie* sample for a command-line tool that creates reference movies.

For more details on reference movies and instructions on how to set them up see "Applications and Examples" in *HTML Scripting Guide for QuickTime*.

Creating a Poster Image for Movies

The video is not decoded until the user enters movie playback mode. Consequently, when displaying a webpage with video, users may see a gray rectangle with a QuickTime logo until they tap the Play button. Therefore, use a poster JPEG as a preview of your movie. If you use QuickTime Pro 7.2.1 or later to export your movies, as described in "Encoding Video for Wi-Fi, 3G, and EDGE" (page 78), then a poster image is already created for you. Otherwise, follow these instructions to set a poster image.

If you are using the `<video>` element, specify a poster image by setting the `poster` attribute as follows:

```
<video poster="poster.jpg" src="movie.m4v" ...> </video>
```

If you are using an `<embed>` HTML element, specify a poster image by setting the image for `src`, the movie for `href`, the media MIME type for `type`, and `myself` as the `target`:

```
<embed src="poster.jpg" href="movie.m4v" type="video/x-m4v" target="myself"
scale="1" ...>
```

Make similar changes if you are using the `<object>` HTML element or JavaScript to embed movies in your webpage.

On the desktop, this image is displayed until the user clicks, at which time the movie is substituted.

Configuring Your Server

HTTP servers hosting media files for iOS must support byte-range requests, which iOS uses to perform random access in media playback. (Byte-range support is also known as content-range or partial-range support.) Most, but not all, HTTP 1.1 servers already support byte-range requests.

If you are not sure whether your media server supports byte-range requests, you can open the Terminal application in Mac OS X and use the `curl` command-line tool to download a short segment from a file on the server:

```
curl --range 0-99 http://example.com/test.mov -o /dev/null
```

If the tool reports that it downloaded 100 bytes, the media server correctly handled the byte-range request. If it downloads the entire file, you may need to update the media server. For more information on `curl`, see *Mac OS X Man Pages*.

Ensure that your HTTP server sends the correct MIME types for the movie filename extensions shown in Table 8-1.

Table 8-1 File name extensions for MIME types

Extensions	MIME type
`.mov`	video/quicktime
`.mp4`	video/mp4
`.m4v`	video/x-m4v
`.3gp`	video/3gpp

Be aware that iOS supports movies larger than 2 GB. However, some older web servers are not able to serve files this large. Apache 2 supports downloading files larger than 2 GB.

RTSP is not supported.

Storing Data on the Client

There are several ways for a web application or website to store data on the client. You can use the JavaScript database classes, described in *Safari Client-Side Storage and Offline Applications Programming Guide*, for storing application data or use the HTML5 application cache for storing resources on the client so webpages continue to display offline when there is no network connection on the desktop and iOS. You can also use the application cache to load webpages faster when there is a slow network connection. This chapter describes how to store data locally using this HTML5 application cache.

To store resources on the client first you create a manifest file specifying which resources to cache. You declare the manifest file in the main HTML file. Then you manipulate the cache and handle related events using JavaScript. Webpages that were previously loaded and contain the resources you specify continue to display correctly when there is no network. The application cache also persists between browser sessions. So, a web application that was previously used on the computer or device can continue to work offline—for example, when iOS has no network or is in airplane mode.

Creating a Manifest File

The manifest file specifies the resources—such as HTML, JavaScript, CSS, and image files —to downloaded and store in the application cache. After the first time a webpage is loaded, the resources specified in the manifest file are obtained from the application cache, not the web server.

The manifest file has the following attributes:

- It must be served with type `text/cache-manifest`.
- The first line must contain the text `CACHE MANIFEST`.
- Subsequent lines may contain URLs for each resource to cache or comments.
- Comments must be on a single line and preceded by the # character.
- The URLs are file paths to resources you want to download and cache locally. The file paths should be relative to the location of the manifest file—similar to file paths used in CSS—or absolute.
- The HTML file that declares the manifest file, described in "Declaring a Manifest File" (page 84), is automatically included in the application cache. You do not need to add it to the manifest file.

For example, Listing 9-1 shows a manifest file that contains URLs to some image resources.

Listing 9-1 Sample manifest file

```
CACHE MANIFEST

demoimages/clownfish.jpg
demoimages/clownfishsmall.jpg
```

```
demoimages/flowingrock.jpg
demoimages/flowingrocksmall.jpg
demoimages/stones.jpg
demoimages/stonessmall.jpg
```

Declaring a Manifest File

After you create a manifest file you need to declare it in the HTML file. You do this by adding a `manifest` attribute to the `<html>` tag as follows:

```
<html manifest="demo.manifest">
```

The argument to the `manifest` attribute is a relative or absolute path to the manifest file.

In most cases, creating a manifest file and declaring it is all you need to do to create an application cache. After doing this, the resources are automatically stored in the cache the first time the webpage is displayed and loaded from the cache by multiple browser sessions thereafter. Read the following sections if you want to manipulate this cache from JavaScript.

Updating the Cache

You can wait for the application cache to update automatically or trigger an update using JavaScript. The application cache automatically updates only if the manifest file changes. It does not automatically update if resources listed in the manifest file change. The manifest file is considered unchanged if it is byte-for-byte the same; therefore, changing the modification date of a manifest file also does not trigger an update. If this is not sufficient for your application, you can update the application cache explicitly using JavaScript.

Note that errors can also occur when updating the application cache. If downloading the manifest file or a resource specified in the manifest file fails, the entire update process fails. If the update process fails, the current application cache is not corrupted—the browser continues to use the previous version of the application cache. If the update is successful, webpages begin using the new cache when they reload.

Use the following JavaScript class to trigger an update to the application cache and check its status. There is one application cache per document represented by an instance of the `DOMApplicationCache` class. The application cache is a property of the `DOMWindow` object.

For example, you get the `DOMApplicationCache` object as follows:

```
cache = window.applicationCache;
```

You can check the status of the application cache as follows:

```
if (window.applicationCache.status == window.applicationCache.UPDATEREADY)...
```

If the application cache is in the `UPDATEREADY` state, then you can update it by sending it the `update()` message as follows:

```
window.applicationCache.update();
```

If the update is successful, swap the old and new caches as follows:

```
window.applicationCache.swapCache();
```

The cache is ready to use when it returns to the `UPDATEREADY` state. See the documentation for `DOMApplicationCache` for other status values. Again, only webpages loaded after an update use the new cache, not webpages that are currently displayed by the browser.

iOS Note: Using JavaScript to add and remove resources from the application cache is currently not supported.

Handling Cache Events

You can also listen for application cache events using JavaScript. Events are sent when the status of the application cache changes or the update process fails. You can register for these events and take the appropriate action.

For example, register for the `updateready` event to be notified when the application cache is ready to be updated. Also, register for the `error` event to take some action if the update process fails—for example, log an error message using the console.

```
cache = window.applicationCache;
cache.addEventListener('updateready', cacheUpdatereadyListener, false);
cache.addEventListener('error', cacheErrorListener, false);
```

See the documentation for `DOMApplicationCache` for a complete list of event types.

Getting Geographic Locations

iOS Note: Geographic location classes are available in iOS 3.0 and later.

Use the JavaScript classes described in this chapter to obtain or track the current geographic location of the host device. These classes hide the implementation details of how the location information is obtained—for example, using Global Positioning System (GPS), IP addresses, Wi-Fi, Bluetooth, or some other technology. The classes allow you to get the current location or get continual updates on the location as it changes.

Geographic Location Classes

The `Navigator` object has a read-only `Geolocation` instance variable. You obtain location information from this `Geolocation` object. The parameters to the `Geolocation` methods that get location information are mostly callbacks, instances of `PositionCallback` or `PositionErrorCallback`. Because there may be a delay in getting location information, it cannot be returned immediately by these methods. The callbacks that you specify are invoked when the location information is obtained or an error occurs. If the location information is obtained, the position callback is passed a position object describing the geographic location. If an error occurs, the error callback is passed an instance of `PositionError` describing the error. The position object represents the location in latitude and longitude coordinates.

Getting the Current Location

The most common use of the `Geolocation` class is to get the current location. For example, your web application can get the current location and display it on a map for the user. Use the `getCurrentPosition` method in `Geolocation` to get the current location from the `Navigator` object. Pass your callback function as the parameter to the `getCurrentPosition` method as follows:

```
// Get the current location
 navigator.geolocation.getCurrentPosition(showMap);
```

Your callback function—the `showMap` function in this example—should take a position object as the parameter as follows:

```
function showMap(position) {
  // Show a map centered at position
}
```

Use the `coords` instance variable of the passed-in position object to obtain the latitude and longitude coordinates as follows:

```
latitude = position.coords.latitude;
longitude = position.coords.longitude;
```

Tracking the Current Location

You can also track the current location. For example, if your web application displays the current location on a map, you can register for location changes and continually scroll the map as the current location changes. When you register for location changes, you receive a callback every time the location changes. The callbacks are continual until you unregister for location changes.

Use the `watchPosition` method in the `Geolocation` class to register for location changes. Pass your callback function as the parameter. In this example, the `scrollMap` function is invoked every time the current location changes:

```
// Register for location changes
var watchId = navigator.geolocation.watchPosition(scrollMap);
```

The callback function should take a position object as the parameter as follows:

```
function scrollMap(position) {
  // Scroll the map to center position
}
```

Similar to "Getting the Current Location" (page 87), use the `coords` instance variable of the passed in position object to obtain the latitude and longitude coordinates.

Use the `clearWatch` method in the `Geolocation` class to unregister for location changes. For example, unregister when the user clicks a button or taps a finger on the map as follows:

```
function buttonClickHandler() {
  // Unregister when the user clicks a button
  navigator.geolocation.clearWatch(watchId);
}
```

Note: Constantly tracking the current location may reduce the device's battery life since the GPS harware is enabled in the tracking mode.

Handling Location Errors

Your web application should handle errors that can occur when requesting location information. For example, display a message to the user if the location cannot be determined due to poor network connectivity or some other error.

When registering for location changes, you can optionally pass an error callback to the `watchPosition` method in the `Geolocation` class as follows:

```
// Register for location changes
var watchId = navigator.geolocation.watchPosition(scrollMap, handleError);
```

The error callback should take a `PositionError` object as the parameter as in:

```
function handleError(error) {
  // Update a div element with the error message
}
```

Debugging

You should test your web content on both the desktop and various iOS devices. If you do not have iOS devices for testing, you can use Simulator in the iOS SDK. Because the screen is larger and Safari behaves slightly differently on iPad, you should specifically test your content on an iPad device or by setting the hardware device in Simulator to iPad. You can also simulate iPad-like behavior in Safari on the desktop by changing the user agent string. When testing in Safari on any platform, you can use the Debug Console to debug your web content.

Safari on iOS provides a Debug Console that allows you to debug your web content and applications in Simulator and the device. The console displays errors, warnings, tips, and logs for HTML, JavaScript, and CSS. The console also shows uncaught JavaScript exceptions as errors. This chapter describes how to enable the Debug Console, view the console messages, and print your own messages to the console.

For more tips on debugging web content in Safari, read *Safari User Guide for Web Developers*. Read the section "Changing the User Agent String" in *Safari User Guide for Web Developers* for how to simulate iPad-like behavior in Safari on the desktop—select Other from the User Agent submenu and enter the iPad user agent string described in "Using the Safari User Agent String" (page 31).

iOS Note: The Debug Console is available in iOS 1.1.1 and later.

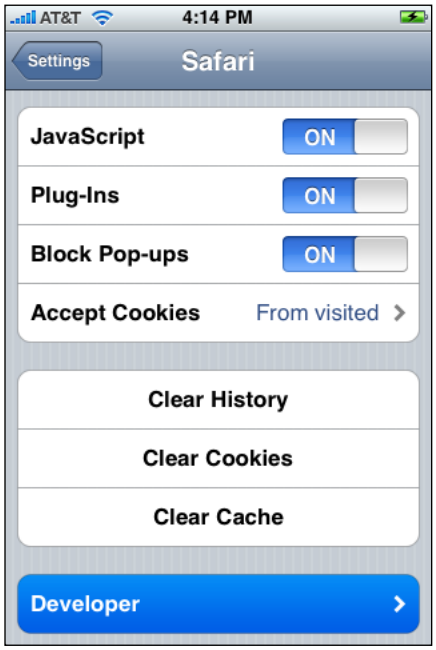
Enabling the Safari Console

You turn on the Debug Console using the Developer Settings for Safari on iOS as follows:

1. Click Settings on the main page.
2. Click Safari.

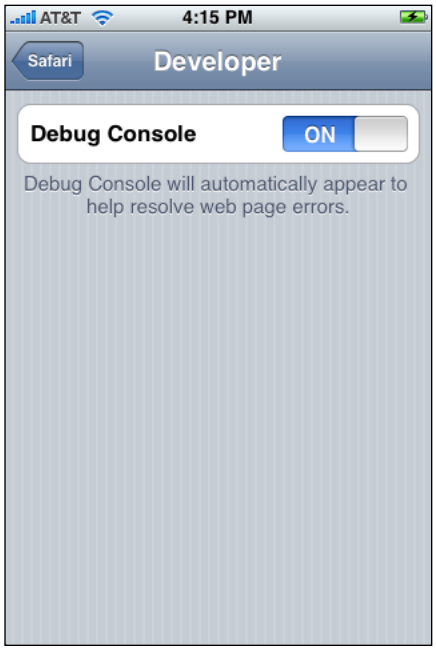
3. Scroll down and select Developer as shown in Figure 11-1.

Figure 11-1 Selecting Developer settings



4. Switch Debug Console to ON as shown in Figure 11-2.

Figure 11-2 Enabling the Debug Console



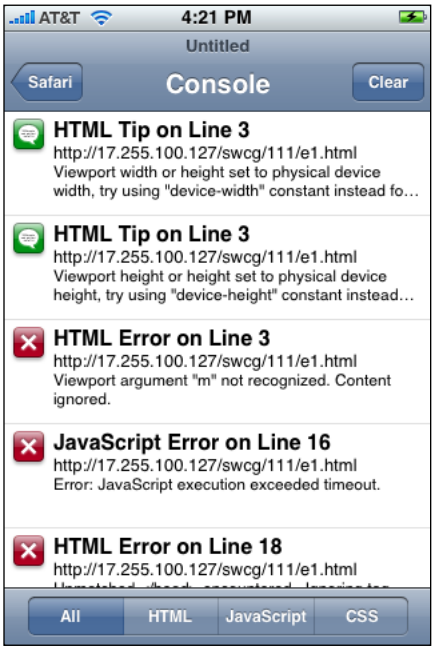
Viewing Console Messages

When the Debug Console is on, a banner appears above your webpage if there are messages in the console, as shown in Figure 11-3. Click the banner to view the messages.

Figure 11-3 The message banner

The console displays the messages in the order in which they occur, as shown in Figure 11-4. Each message description contains the type of message, the line number and filename of the document, and the text message. The types of messages are log, info, warning, and error. Click the Clear button to remove all messages from the console.

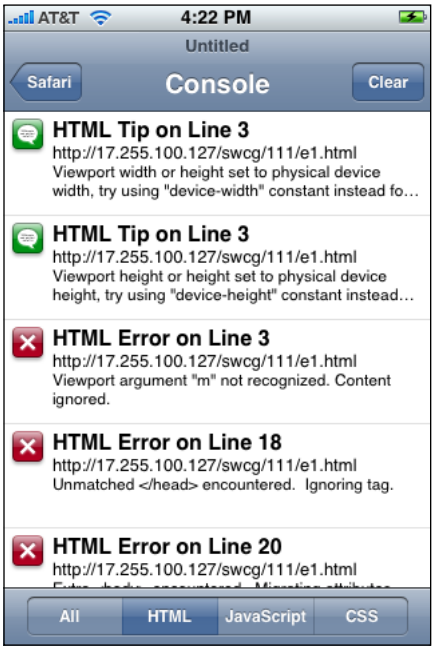
Figure 11-4 Messages in the console



You can also filter the list by HTML, JavaScript, and CSS messages by clicking the respective button at the bottom of the screen. Figure 11-5 shows the HTML messages filtered on the left and the JavaScript messages filtered on the right.

Figure 11-5 Filtered console messages

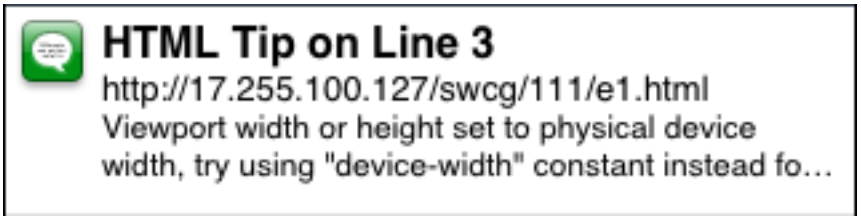
HTML messages



JavaScript messages

Tips appear in the console if you don't follow certain guidelines. For example, if you set the viewport width to 320 instead of using the `device-width` constant, then an HTML tip appears in the console as shown in Figure 11-6. A similar tip appears if you set the viewport height to 480 instead of using the `device-height` constant.

Figure 11-6 Viewport width or height tip



For example, the following HTML fragment generates the first three HTML messages in the console shown on the left side in Figure 11-5 (page 92). Syntax errors also generate messages.

```
<meta name = "viewport" content = "width = 320, height = 480, m = no">
```

The console catches JavaScript errors too. For example, the following JavaScript code causes a timeout—it contains an infinite loop—and generates the error message shown in Figure 11-7.

```
<script>
while (true) a = 9;
</script>
```

Figure 11-7 JavaScript timeout message

Creating Messages in JavaScript

You can print your own messages to the console using JavaScript. You do this by using the `console` attribute of the `window` object in JavaScript.

For example, the following code line prints the value of a variable to the console using the `log()` method:

```
console.log("x = " + x);
```

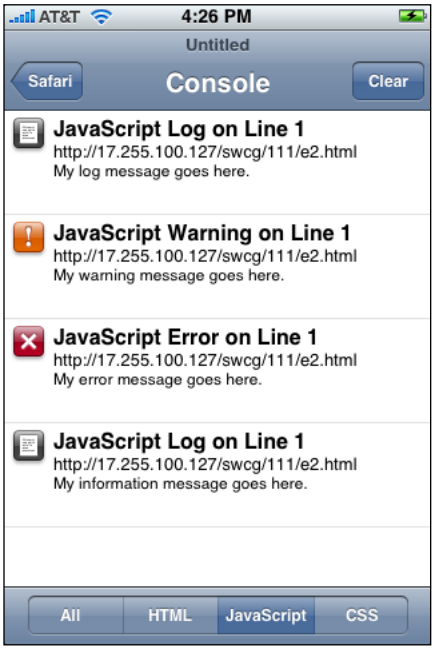
You can also create your own error and warning messages using the `error()` and `warn()` methods as in this code fragment:

```
<script>
console.log("My log message goes here.");
console.warn("My warning message goes here.");
```

```
console.error("My error message goes here.");
console.info("My information message goes here.");
</script>
```

This code fragment generates the console messages shown in Figure 11-8. The `info()` method is the same as `log()`—it's provided for backward compatibility.

Figure 11-8 Console messages from your JavaScript code



HTML Basics

HyperText Markup Language (HTML) is the fundamental mark-up language used to create web content. Your HTML needs to be well structured and valid to work well with Safari on the desktop and Safari on iOS. Read this appendix to learn more about creating conforming HTML content.

See *Safari HTML Reference* for a complete guide to all the HTML elements supported by Safari.

What Is HTML?

HTML is the standard for content structure on the web. Its original intention of the designers was to provide the structure required for web browsers to parse its content into a meaningful format. This structure could define entire documents, complete with headings, text, lists, data tables, images, and more. As the web flourished, it also began to incorporate style and multimedia aspects as well.

Arguably the most important feature of HTML is the ability to "hyperlink" text. This gives content providers the ability to assign the URI of other content on the web to a block of text, allowing it to be clicked and followed by the user of the content.

The most recent revisions of the HTML standard are returning to the "old days" of separating the structure of web content (HTML) from the presentation of the content (using a technology called Cascading Style Sheets, or CSS). You can learn more about creating effective web content style in the "CSS Basics" (page 101) appendix.

This appendix, conversely, covers only the structure of HTML and how to properly format a document for a variety of clients. It does not discuss advanced HTML features or proper webpage layout and design.

Basic HTML Structure

There are a few basic structure blocks that make up the core of an HTML document. The blocks are described in the context of the HTML code shown in Listing A-1.

Listing A-1 Basic HTML document

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
    "http://www.w3.org/TR/REC-html40/strict.dtd">
<html>
    <head>
        <title>HTML Sample Code</title>
    </head>
    <body>
        <div>
            <img src="myWelcomeGraphic.gif" alt="Welcome!" />
        </div>
```

```
        <h1>Big Heading</h1>
        <p>This is our HTML sample code. It shows many elements:</p>
        <ul>
            <li>The HTML document block.</li>
            <li>The HEAD and title of the page.</li>
            <li>A paragraph.</li>
            <li>An unordered list.</li>
        </ul>
    </body>
</html>
```

The html document block: The `<html>` document block is the entirety of the HTML code for a webpage. In the example, the tags defining this block—`<html>` and `</html>`—are located towards the top and bottom of the document. The document is prefaced with a `DOCTYPE` declaration, which tells browsers which specification to parse your webpage against. If you are following the strict conventions of the HTML specification, you should use the declaration shown above. Otherwise it can be left off, but it defaults to a "quirks" mode. Refer to the HTML 4.01 Specification for more on document validation types.

The head block: The `<head>` block defines a block of metadata about the webpage. In this case, you can see the webpage has a `<title>` element within it. The title is the text that is displayed at the top of a web browser window. The `<head>` block also can contain a variety of other metadata, such as externally linked CSS style sheets (using the `link` tag) and sets of JavaScript functions. This block should always contain at least the title, and should always be external to the body content.

The body block: This block defines the entire body of the document—it should encompass the visible content of the webpage itself. The body block itself is not designed for inline content. Rather, you should define other block elements (such as paragraphs, divisions, and headers) and embed content within them. The `<body>` block should be used to specify style parameters for the entirety of the content.

Other block elements: There are a number of other fundamental block elements enclosed within the content's `<body>` block. They include:

- **Heading.** Specified in this case by the `<h1>` and `</h1>` tags, this defines the header for a following block of content. The headers can be of six different sizes, ranging from a very large first-level heading (defined with the `<h1>` and `</h1>` tags) down to a small sixth-level heading (defined with the `<h6>` and `</h6>` tags). It should contain only brief text—other content such as large text blocks, images, and movies should be embedded in other appropriate block elements such as paragraphs and divisions.

- **Paragraph.** Specified by the `<p>` and `</p>` tags, this is one of the fundamental block elements for web content. Each individual paragraph should contain the inline text content that defines the readable content of a webpage and should not enclose any other block elements. Generally, paragraph blocks are for text only. An alternative to the paragraph is the division, and that is the most appropriate block element for other media types such as images and movies.

- **Division/Section.** Specified by the `<div>` and `</div>` tags, the division is designed to contain all kinds of content, including text, images, and other multimedia. It also can encompass other block elements such as paragraphs, though enclosing divisions within other divisions is generally not recommended. Generally, division blocks are used to define unified styles for blocks of content. In the example above, the division block contains the heading image for the webpage.

- **List.** HTML supports two basic kinds of lists, the ordered list (specified by the `<ol>` and `</ol>` tags) and the unordered list (specified by the `<ul>` and `</ul>` tags), as in the example above. An ordered list tags each list element (specified by the `<li>` and `</li>` tags) with an incremental number (1, 2, 3, and so on). An unordered list tags each list element with a bullet, though this marker can be changed using CSS styling.

Now you've learned some of the fundamental skeleton elements of HTML structure. Block elements such as paragraphs and divisions are the core of the content—by themselves they are invisible, but they contain inline elements such as text, images, and movies. The next section takes you a little deeper into some features of HTML content.

Creating Effective HTML Content

You've learned about the fundamental elements that define HTML structure, but a webpage is useless without any kind of content in it. Now that you've laid down the foundation for the webpage, you should place some content to create a rich experience for your users. This appendix discusses some basic inline HTML elements; for all the elements supported by Safari and WebKit, refer to *Safari HTML Reference*.

The most common web content contains a lot of text and a few images. Think of a travel journal, for example, that has a discussion of the day's events alongside a few photos from the journey. As the Internet has matured, you may have seen more in the way of movies, animations, and other "rich" forms of content introduced to the web. But the most common media is still a combination of text and images.

Displaying text is a simple thing in HTML. Once you've established the surrounding block element—a paragraph, for example, as discussed in the previous section—the text can just be placed inline. An example from the fictional travel journal might be as shown in Listing A-2:

Listing A-2 Adding a paragraph

```
<p>
Today, we arrived in Cupertino, California. We visited the Apple campus. It was
 a bright sunny day and exhibited none of the fog that was so prevalent during
 our stay in San Francisco.
</p>
```

It's a simple textual entry, but there's not much else to it. A good travel journal also marks the date and time of each entry, so you should add that to the content, as well. Listing A-3 shows the time and date added as a heading.

Listing A-3 Adding a heading

```
<h1>Friday, May 20, 2005 - 4:40PM</H1>
<p>
Today, we arrived in Cupertino, California. We visited the Apple campus. It was
 a bright sunny day and exhibited none of the fog that was so prevalent during
 our stay in San Francisco.
</p>
```

It's still a simple textual entry, but at least you've provided your reader with a little extra information. But what if your reader has no idea what Apple is? One of the great features of HTML is the ability to "hyperlink" documents—create links to external webpages. Using the `<a>` and `</a>` hyperlink tags, you can link your reader to the Apple website as shown in Listing A-4.

Listing A-4 Creating a hyperlink

```
<h1>Friday, May 20, 2005 - 4:40PM</H1>
<p>
Today, we arrived in Cupertino, California. We visited the <a
href="http://www.apple.com/" title="Apple web site">Apple</a> campus. It was a
```

```
 bright sunny day and exhibited none of the fog that was so prevalent during
our stay in San Francisco.
</p>
```

Notice that the word "Apple" is now surrounded by this hyperlink element. The element describes two particular attributes:

- **The href attribute:** This links to the URL of the webpage you want to link to. If you specified a relative URL, such as "myPage.html" or "/pages/myPage.html", the link would point to a file within the same folder as your code, or in a separate folder, respectively. In this example, the value is a fully qualified URL, so it simply links to that site (the Apple homepage).
- **The title attribute:** This is an optional attribute, but one you should get into the habit of using. The `title` attribute provides an alternate description of the link. In Safari, holding the mouse over the hyperlink for a couple of seconds reveals this value as a tooltip. It's a great way to provide information about a link before the user clicks it, letting them decide if they want to leave your webpage or not. Additionally, this information is used by screen readers and other accessibility devices, so by using this attribute, you help extend your content to a larger community.

With this hyperlink in place, the word "Apple" in the travel journal is now displayed as a clickable link. Clicking the word redirects the user to the Apple homepage.

So far the travel journal reads great. But to really capture the attention of your readers, you might want to include an image. An image in HTML is specified by the `<img>` tag. It's important to note that an image is an inline element, so needs to be placed within a block element such as a paragraph. It is also a little different from some other inline elements in that it doesn't require a closing tag. Listing A-5 shows how to add an image to the travel journal entry.

Listing A-5 Adding an image

```
<h1>Friday, May 20, 2005 - 4:40PM</h1>
<p>
<IMG src="cupertino.gif" alt="Picture of Cupertino, CA" />
<br />
Today, we arrived in Cupertino, California. We visited the <a
href="http://www.apple.com/" title="Apple web site">Apple</a> campus. It was a
 bright sunny day and exhibited none of the fog that was so prevalent during
our stay in San Francisco.
</p>
```

Notice that the image definition looks a lot like the hyperlink definition. The `src` attribute defines the URL to the image (with the same rules for relative versus absolute URLs as in the hyperlink), and the `alt` attribute defines a block of alternate text—this text can also be read by screen readers, or can be shown by some browsers when images are turned off in the browser.

Another small element we added was the `<br>` line break element. Remember that an image is an inline element, just like text. Without a forced line break, the image would display and the text would follow directly after, left to right, one after the other. That's a little awkward for a travel journal, but useful when you have small images (like mathematical equations) that you want integrated into the text. Add the line break to force the next line of text to a new line.

Now you've learned about actual web content—the inline text and media that defines what a user reads and views when they visit your webpage. This section is by no means an exhaustive discussion on the content you can provide to your users. For more information on the content that Safari and WebKit support, refer to *Safari HTML Reference*.

Using Other HTML Features

This section discusses a few more features of HTML that you may want to use in your web content.

One other common block element is the `<table>` block. You can add a `<table>` block to display any kind of tabular data. To the previous example, let's add a table of temperatures that the journal writer experienced on his or her day in Cupertino. For the information to be useful, you'll also want to add something about the time at which the temperature was recorded. Both the time and temperature can be labeled using table headers, specified by the `<th>` and `</th>` tags. Notice that the order of the table headers and table cells (specified by the `<TD>` and `</TD>` tags) match within their particular row (specified by the `<tr>` and `</tr>` tags) in Listing A-6.

Listing A-6 Creating a table

```
<h1>Friday, May 20, 2005 - 4:40PM</h1>
<p>
<img src="cupertino.gif" alt="Picture of Cupertino, CA" />
<br />
Today, we arrived in Cupertino, California. We visited the <a
href="http://www.apple.com/" title="Apple web site">Apple</a> campus. It was a
 bright sunny day and exhibited none of the fog that was so prevalent during
our stay in San Francisco.
</p>

<table border="1" cellpadding="5" cellspacing="5">
    <tr>
        <th>Time</th>
        <th>Temperature</th>
    </tr>
    <tr>
        <td>9:00AM</td>
        <td>65 degrees</td>
    </tr>
    <tr>
        <td>12:00PM</td>
        <td>76 degrees</td>
    </tr>
    <tr>
        <td>3:00PM</td>
        <td>78 degrees</td>
    </tr>
</table>
```

The table definition itself contains some special attributes. The `border` attribute defines the width of the border surrounding the table. The `cellpadding` attribute defines the width of the space between the cell border and the content within. The `cellspacing` attribute defines the width of the spacing between individual cells.

Another useful feature is the ability to integrate JavaScript—an interpreted language processed by web browsers—within HTML. JavaScript can do a variety of tasks, many of which are addressed in *WebKit DOM Programming Topics*. The JavaScript code can be embedded in external files, within the `<script>` block of the webpage's `<head>` block, or even inline with the elements, using the various JavaScript delegates provided by the browser. For example, if you want to display an alert when the user clicks a button, add the code (or the function call, if the code is defined elsewhere) to the button's `onClick` delegate:

```
<input type="button" value="Click Me!" onClick="alert('This is an alert!')">
```

If you are new to JavaScript, read *Apple JavaScript Coding Guidelines*.

CSS Basics

Cascading Style Sheets (CSS) separates the presentation details from the HTML content, allowing you to create style sheets for different platforms. If you are optimizing your web content for Safari on iOS, you need to use CSS to access some of the iOS web content features. Read this appendix to learn how to add CSS to existing HTML content.

See *Safari CSS Reference* for a complete guide to all the CSS properties supported by Safari.

What Is CSS?

CSS is an extension to standard HTML content that allows you to fine-tune the presentation of web content. With CSS you can change a variety of style attributes of the content you are designing, such as the font for a block of text, the background color of a table, or the leading (line spacing) between lines of text.

CSS allows you to cater to different clients and preferences, because you can change the style of a webpage on the fly without ever editing the HTML structure. Instead of embedding style within the HTML structure, such as using the `bgcolor` attribute for the webpage body, you should place CSS style definitions in a separate block outside of it. In fact, your webpages are more maintainable if you separate your HTML and CSS code into different files. This way, you can use one style sheet (which holds your style definitions) across multiple webpages, dramatically simplifying your code.

The various ways you can define style for an HTML element within your webpages are described in the remaining sections of this appendix.

Inline CSS

Using inline CSS—where style definitions are written directly into the HTML element definition—is perhaps the easiest way to define style for an element. You can do this using the `style` attribute for the element. For example, start with this paragraph:

```
<p>The quick brown fox jumped over the lazy dog.</p>
```

Without any style definitions, this renders in the default paragraph font and style for the browser rendering it. But let's say you wanted to change the style of the paragraph to display in a boldface. You can do this with the CSS `font-weight` property. To change the style for this one paragraph, add the `font-weight` key and the value for the style you want directly to the paragraph's `style` attribute:

```
<p style="font-weight: bold;">The quick brown fox jumped over the lazy dog.</p>
```

This changes the font style of that paragraph to boldface. There are some downsides to using the style definitions inline with the HTML, though:

- The definition is not reusable. For each paragraph that you want displayed in boldface, you have to type the same style definition—one for each paragraph. If you wanted to change the bold style to an italic style, for example, you would have to change the definition for each and every paragraph, as well.
- The code can get cluttered. Most of the time, you won't have a single style definition. For a particular paragraph, you may want to have it display in boldface, indent it 20 pixels from the left margin, and give it a blue background color with a black border. At minimum, this requires four CSS style definitions *for each paragraph you want to match this style.*

One of the big advantages of CSS is the ability to separate the style from the structure, but that advantage is lost with this method. Other methods of using CSS in your content preserve the advantage, as explained in the following sections.

Head-Embedded CSS

Near the beginning of every HTML document is a `<head>` block, which defines invisible metadata about the content. Within this section you can define a variety of CSS definitions that you can then reuse within the actual body content.

In the previous section there was an example of a paragraph in boldface with a blue background and a black border, all indented 20 pixels from the left margin. The definitions for that style look like this:

```
font-weight: bold;
background-color: blue;
border: 1px solid black;
margin-left: 20px;
```

But how do you embed these definitions within HTML elements without typing them directly into the HTML? First, you need to define them within the style section of the `<head>` block for the webpage. Second, you need an identifier to isolate that particular set of style definitions from any others in the `<style>` block. Using the identifier `notebox`, the style definition looks like this:

```
...
<head>
    <style type="text/css">
        .notebox {
            font-weight: bold;
            background-color: blue;
            border: 1px solid black;
            margin-left: 20px;
         }
    </style>
</head>
...
```

Notice that the definitions are bound by braces, and that the identifier (`notebox`) is preceded by a period. The latter allows you to use this set of style definitions for *any* element within your HTML content. If you want to limit its use only to paragraph elements, change the identifier to:

```
P.notebox
```

This tells the browser to use the definitions only if they are defined within a `<p>` paragraph element. If you want to use these styles for *all* paragraphs, then you don't need the custom identifier. Change the identifier to `p`.

You've learned how to define the custom styles in the `<head>` block of your content. But how do you actually tell the browser which paragraphs should use these styles? Here are two paragraphs of text in HTML:

```
<p>This is some plain boring text.</p>
<p class="notebox">This is a finely styled paragraph!</p>
```

There's a new attribute in the second paragraph: `class`. This is how you specify the style definition that a particular element should render itself with. The top paragraph in the example above would render as usual, in the default paragraph style for the browser. But with the style class of the second paragraph set to your new `notebox`, it will render with a bold font, a blue background color, a 1-pixel solid black border, all 20 pixels from the left margin. For any paragraph (or any element, since we didn't specify an explicit element it could be assigned to), simply use that class attribute to name the identifier of your style definition.

There is however one disadvantage to this method of embedding CSS in a webpage. Though the definitions are reusable within the webpage—you can now specify as many `notebox` paragraphs as you want—they are *not* reusable across multiple webpages. If you want the paragraph's text to be rendered in an italic style instead of a bold one, you'd have to change that definition on each webpage where you integrated it. The next section describes the most scalable way to use CSS within your web content.

External CSS

If you want to use a particular style across multiple webpages, there's only one way to do it: externally linked style sheets. Since each webpage has to know about the style definitions you created, placing all of them into an external file and then linking each webpage to that file seems like a reasonable way to inform them. That way, if you want to change boldface to italic, you only have to change it once—in the external file.

An external style sheet is almost exactly the same as the `<style>` block that you defined in the last section, but it's not embedded in HTML. All the browser needs are the style definitions themselves. Listing B-1 shows a new file, called `styles.css`, that contains all the style definitions for the webpage.

Listing B-1 The styles.css file

```
.bordered {
    font-weight: bold;
    background-color: blue;
    border: 1px solid black;
    margin-left: 20px;
}

.emphasized {
    font-style: italic;
}
```

For good measure, there is another style definition—one that simply sets the font style of the element's text to an italic style. Now you have to somehow let the HTML content know about this external style sheet. You won't have any more embedded style definitions, so you can remove the `<style>` block altogether. In its place—still in the `<head>` block of the webpage—you'll add a `<link>` element that links the external style sheet to the document:

```
<link rel="stylesheet" href="styles.css" type="text/css">
```

This line tells the browser to link to this external style sheet. Note that the URL specified by `href` is relative—for this particular line to link the style sheet correctly, `styles.css` must be in the same folder as the HTML linking to it.

Once you've included this line, you can use the HTML `class` attribute just as in the previous section:

```
<p>This is some plain boring text.</p>
<p class="emphasized">This is some italic text.</p>
<p class="bordered">This is a finely styled paragraph!</p>
```

You've learned how to integrate CSS style into your web content. For information on what kinds of CSS properties and features are supported by Safari and WebKit, refer to *Safari CSS Reference*.

Document Revision History

This table describes the changes to *Safari Web Content Guide*.

Date	Notes
2010-12-16	Applied minor edits.
2010-11-15	Updated per iOS 4.2 changes.
2010-08-23	Applied minor edits throughout.
2010-03-24	Made changes related to iPad throughout.
2010-01-20	Minor edits.
2009-09-09	Updated the iOS resource limits again.
2009-08-11	Updated the iOS resource limits.
2009-06-08	Changed the title from "Safari Web Content Guide for iPhone OS" and applied minor edits throughout.
2009-03-05	Minor edits throughout.
2009-01-30	Minor edits throughout.
2009-01-06	Moved appendix to separate book called Apple URL Scheme Reference. Removed redundant reference now included in the Safari HTML Reference and Safari DOM Extensions Reference books.
2008-11-17	Added the chapter "Storing Data on the Client."
2008-10-15	Minor edits throughout.
2008-09-09	Updated for iOS 2.1.
2008-07-15	Updated for iOS 2.0.
2008-02-05	Updated book link in "Specifying a Webpage Icon for Web Clip".
2008-01-15	Added section on specifying a web clip icon.
2007-10-31	Added instructions for exporting movies for iPhone using Quicktime Pro 7.2.1.
2007-10-11	Added figures to the "Customizing Style Sheets" and "Debugging" articles. Removed the "Configuring Keyboard" section from "Designing Forms" because using the lang property to select keyboard languages is deprecated.

Document Revision History

Date	Notes
2007-09-27	Changed the title from Safari Web Content Guide. Completely revised to describe how to create web content for Safari on the desktop and Safari on iPhone using Web 2.0 technologies.
2005-08-11	Corrected typos.
2005-06-04	New document that discusses creating effective web content for Safari and the Web Kit.