

CHAPTER 8

HTML5 Forms

Forms are a key issue on mobile devices. While some people claim that typing on mobile devices is painful, we can see how much typing happens today on mobile devices: from SMSs to WhatsApp, Twitter, and Facebook. Even if typing on these devices is not as comfortable as on desktops, users are doing a lot of it, and it's our job to make it as good an experience as possible.

To maximize compatibility and accessibility, the input controls should be inside the classic form tag, with `method="GET"` or `"POST"` and `action="serverScriptURL"`. If we want to use Ajax or any other sort of rich behavior, that's fine, but we must be sure to always have a basic HTML form as a fallback.

Form Design

Avoid using tables for form layout. The best solution is to use definition lists, labels, and input controls. Using CSS you can enhance the form and even with different layouts for different devices and orientations. If you start your design with a fixed-layout from the HTML, such as using tables, you will not have the ability to modify it from CSS.



The older BlackBerry browser for smartphones—not BB10—allowed offline form submission. If the device was offline when the user completed the form, it was placed in a queue and was automatically submitted when the device went back online.

A typical key/value form should look like the following code:

```
<form action="formAction" method="post">
  <dl>
    <dt><label for="name">Name</label></dt>
    <dd><input type="text" name="name"></dd>
```

```
</dl>
</form>
```

The usage of the `label` tag is very important for mobile input controls, and especially for touch devices. For example, if you insert a checkbox without a `label` tag, the user will need to tap (click) over the tiny checkbox to select it, which can be fiddly. Using a `label` allows the user to tap anywhere in the text assigned to the checkbox to select it.



As discussed in Chapter 5, mobile forms should have a vertical design, so it is better to put the label above each input field instead of to the right or left. This is because touch devices zoom in on the field when it's in focus and do not show what is to the right or left of the control. Tables don't help in this design.

So, a form with a checkbox should look something like the following:

```
<form action="formAction" method="post">
  <input type="checkbox" name="accept" id="accept" value="yes">
  <label for="accept">I accept terms and conditions</label>
</form>
```



Some mobile browsers have a lower limit for the URL length than desktop ones. That is why we should avoid long forms using the GET method.

We can also assign access keys to the form controls (using the `accesskey` attribute in the `input` tags) and show which keys are assigned in the labels, with a CSS class. This method is very useful in devices with QWERTY keyboards, where you can assign a letter to each field instead of numeric values:

```
<form action="formAction" method="post">
  <input type="checkbox" name="accept" id="accept" value="yes"
    accesskey="a">
  <label for="accept">I <span class="accesskey">A</span>cept terms
    and conditions</label>
</form>
```

A typical form should include one or more `fieldset` tags, each with a `legend` inside. The `fieldset` is just a container for form controls, and the `legend` is a child tag that defines the title or legend for its parent:

```
<fieldset>
  <legend>Personal Information</legend>
  <!-- controls here -->
</fieldset>
```

Some mobile browsers will add navigation between all interactive form elements, such as Previous or Next buttons (Figure 8-1) on virtual keyboards that will jump between elements in the current order, or `tabindex` declarations, as if it were tabbed navigation on a desktop. Therefore, it's important to define a coherent order for the elements.



Figure 8-1. Some browsers allow tabular navigation between form controls using *Previous* and *Next* buttons



To reduce submissions you should add JavaScript form validation, and to secure your app you must add a validation process on the server side.

Form Elements

An HTML5 mobile form can contain the following form elements:

- Single-line text boxes for different types of input
- Password boxes
- Multiline text boxes
- Date and time selectors
- Select lists for single selection
- Select lists for multiple selection
- Checkboxes
- Radio buttons
- File selectors
- Range selectors
- Noninteractive elements, such as hidden input fields and keygen elements

As we are going to see later, not every platform supports every form type, so we need to be careful about which ones we use.

Select Lists

The `select` tag should be one of the most-used tags in a mobile form. Selection from a list is the first option for reducing typing. In a mobile browser, when you click on a select element, typically you will see a pop-up window (modal or not) showing all the options. As shown in Figures 8-2 and 8-3, how select lists are rendered varies across devices.



Figure 8-2. Most mobile browsers show nice native dialogs when selecting elements using standard form controls.

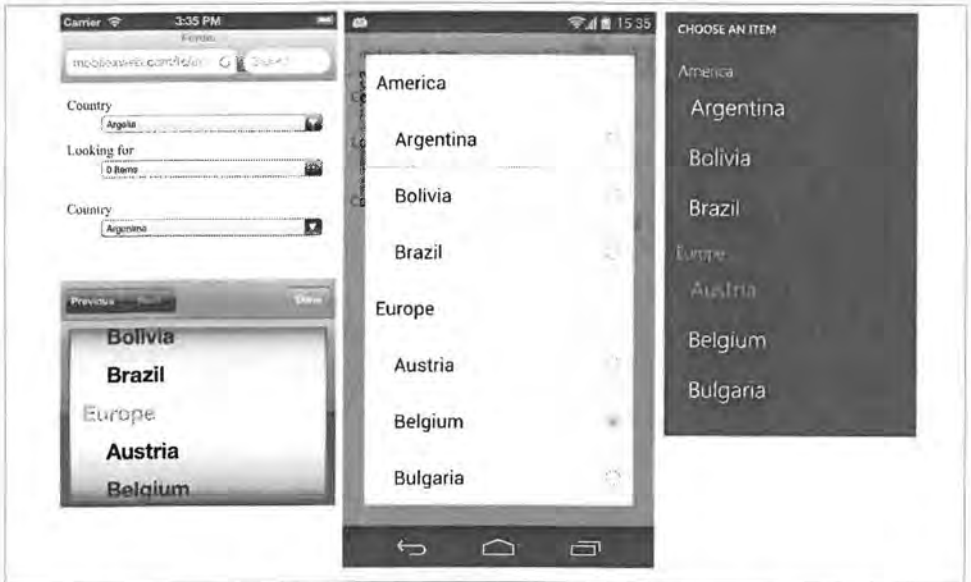


Figure 8-3. Symbian select lists offer a search box for all the options, Android shows a modal pop-up, and Safari uses a wheel selector.

You can use the `size` property of the `select` tag to define a list with a predefined height, but the result is not usually good enough on mobile browsers.

You can specify that the list accepts multiple selection using the Boolean attribute `multiple`. The multiple-selection feature is more useful in mobile forms than desktop forms. In a desktop form, the user generally uses Shift or Control to select multiple options. In a mobile form, we generally present a pop-up window with checkboxes for the user to make his selections and a confirm action to go back to the main page, as seen in Figure 8-2.



With HTML5, Boolean attributes such as `multiple` don't need to have a value definition. If you want to force a value or if you are validating against an XHTML DOCTYPE, the value must be the same name as the attribute name, such as `multiple="multiple"`.

The code for the select lists shown in Figure 8-3 looks like the following:

```
<form action="formAction" method="post">
  <dl>
    <dt><label for="country">Country</label></dt>
    <dd>
      <select name="country">
        <option>Argelia</option>
        <option>Argentina</option>
        <option>Bolivia</option>
        <option>Brazil</option>
      </select>
    </dd>
    <dt><label for="filter">Looking for</label></dt>
    <dd>
      <select name="filter" multiple>
        <option>Flights</option>
        <option>Hotels</option>
        <option>Restaurants</option>
        <option>Car Rental</option>
      </select>
    </dd>
  </dl>
</form>
```



Safari for iOS uses a spinning wheel for selections in HTML, but the native similar control also supports a multicolumn spinning wheel to select multiple fields at the same time. This functionality is not provided in HTML and can only be used natively in Objective-C. However, there is a JavaScript library that emulates the multicolumn control available at www.mobilexweb.com/go/wheel.

Option groups

Option groups are an underused feature of select lists, even in desktop web development. Defining an `optgroup` allows you to provide a label for a set of children, so you can group the available options by category. Here's an example:

```
<dl>
  <dt><label for="country">Country</label></dt>
  <dd>
    <select name="country">
      <optgroup label="America">
        <option value="ar">Argentina</option>
        <option value="bo">Bolivia</option>
        <option value="br">Brazil</option>
      </optgroup>
      <optgroup label="Europe">
        <option value="at">Austria</option>
        <option value="be">Belgium</option>
        <option value="bg">Bulgaria</option>
      </optgroup>
    </select>
  </dd>
</dl>
```

The result is shown in Figure 8-4. Again, different devices may render option groups differently!

Radio Buttons and Checkboxes

The usage of radio buttons and checkboxes is the same on mobile devices as on desktops. The only recommendation I can give you is to avoid using these controls if there are more than four options, using in that case a `select` with single or multiple selection, as appropriate. A group of more than four radio buttons is likely to increase the page height and may require scrolling, which can impact the usability of the form.

Using a label attached to a radio button or a checkbox is a must, as the little icon (the box or circle) is too small for a touch experience.



Figure 8-4. Option groups can be rendered strangely on older devices, but on devices shipped after 2011 they usually work well.

Buttons

HTML has five types of buttons:

Image map buttons

```
<input type="image">
```

Submit buttons

```
<input type="submit">
```

Clear buttons

```
<input type="reset">
```

Custom buttons

```
<input type="button">
```

Submit buttons with HTML support

```
<button></button>
```

The image map button allows us to use an image as a button and receive on the server the coordinates of the point inside the image where the user clicked. Of course, this functionality is possible only on mobile devices supporting touch- or cursor-based

navigation. This type of button should be avoided if possible (this is the recommendation of the W3C) and replaced with a classic submit button. We can later add an image or icon using CSS.

The submit button is the most widely compatible, and for the lowest common denominator devices it should work fine. I've always hated (yes, hated!) the clear button. How many times have you clicked a clear button thinking it was the submit button? This button should be avoided when developing for mobile devices: why potentially add more scrolling and take up more space for a function that few people use? If you want to include this functionality in your form, please be sure to use a different style for the clear button (smaller, darker) than the style you use for the submit button.

The custom buttons should be avoided if you want full mobile compatibility because they work only with JavaScript. For compatible devices, you can still use submit buttons and capture the submit action with JavaScript.

Hidden Fields

Hidden fields (`<input type="hidden">`) are fully compatible with mobile browsers. They are useful for passing information from one page to another because any hidden field will be sent to the server as if the user has entered it on the form. Thanks to modern server-side techniques, hidden fields are not as important today as they once were.

Text Input Fields

Text input on mobile devices is a key area for improving usability.

QWERTY devices are the only ones that don't require a new window to insert the text: when the text input has focus, the user can start typing. In all other device types, when the field has focus the user can either click (or tap) in it or start typing to open a modal pop-up window with all of the OS's typical text typing features (such as predictive text, onscreen keyboard, character recognition, dictionary, and symbol list). Some devices show a full-screen text input window (hiding the entire web interface), and others show a smaller window inside the browser window. In the first case, a descriptive label is required so the user knows what to type.

We can divide the possible text inputs into:

- Single-line inputs
- Multiline inputs
- Rich inputs
- Pop-up inputs



Before iOS 3.0, an undocumented feature allowed us to force a numeric keyboard using the `name` property of the text input. If the name of the input contained the string “zip” or “phone,” the keyboard would change to numeric. However, using this feature was not good practice because it’s inconvenient in non-US countries where the zip code can contain letters (the United Kingdom and Argentina, for example).

For the single-line inputs, HTML5 uses the standard `input` element with different `type` values. The type will determine which keyboard to show on the screen on touch devices. For nontouch devices, every type will react in the exact same way. The possible input types we have are:

- Any-character textbox: `<input type="text">` or just `<input>`
- Numeric textbox: `<input type="number">`
- Phone number textbox: `<input type="tel">`
- Email textbox: `<input type="email">`
- URL textbox: `<input type="url">`
- Any-character secure box: `<input type="password">`
- Search textbox: `<input type="search">`
- Date textboxes: to be covered later in this chapter

As we can see in Figure 8-5, Safari on iOS, the Android browser, and the BlackBerry browser for PlayBook and BB10 will show different virtual keyboards for each input type. For example, there is an email-optimized keyboard that includes the @ character, and a URL-optimized keyboard with a “.com” button and no space bar. In some other browsers, every nonsecure input type will be treated as a `type="text"`.



Figure 8-5. Different virtual keyboards displayed on iOS, Android, and BlackBerry 10 devices—in order: text, email, url, number, and tel.



Even when you are using a numeric input type, JavaScript always treats the value as a string, and on some devices the user can switch to an alphanumeric keyboard and enter nonnumeric characters. Therefore, input constraints must not be used as a validation mechanism, just a feature that will improve usability on some devices.

The search box is just a text box with a different semantic meaning; it lets external search engines know that your site search engine is there. In some browsers, such as Safari on iOS, the search input type is shown with a different UI and the virtual keyboard will use “Search” instead of “Go” on the submit button, as you can see in Figure 8-6.

Some browsers also support the new autosave attribute, which works only with `type="search"` input types. By defining that attribute with a unique ID, we allow the browser to save a user’s searches for future suggestions. We can define how many entries we want to save using the `results` numeric attribute, as in:

```
<input type="search" autosave="mysearch" results="5">
```



Figure 8-6. When using `type=search`, some virtual keyboards will replace the default “Go” button with a search button.

Password or No Password

The usage of the password text input (`<input type="password">`) on mobile devices is a subject of much debate. The password text input (with the classic stars or circles displayed instead of the typed characters) was originally created because of the possibility of a password or other sensitive data being stolen by someone standing behind the user with a view of the screen.

In the mobile ecosystem, the situation is different. With the limited screen size and font size, it is very difficult for another person to see what the user is typing on his mobile phone. Furthermore, typing on non-QWERTY devices is difficult, and if we show a star instead of the real character typed the user may be unsure that he’s entered the text correctly (even if, as is done on some devices, the character is displayed for a second before it’s changed to a star).

Jakob Nielsen (<http://useit.com>), guru of web usability, agrees. In a 2009 Alertbox column, Nielsen wrote: “Usability suffers when users type in passwords and the only feedback they get is a row of bullets. Typically, masking passwords doesn’t even increase security, but it does cost you business due to login failures.”

A good solution to this problem is to add a checkbox to make it visible (changing `type="password"` to `type="text"`), or leave it as `type="text"` and provide a checkbox to make it hidden. Another solution, used by many websites (such as Facebook), is to offer a `type="password"` field on the first login page. If the login details provided are incorrect, it automatically changes to `type="text"` so you can see what you are typing, as you can see in Figure 8-7.

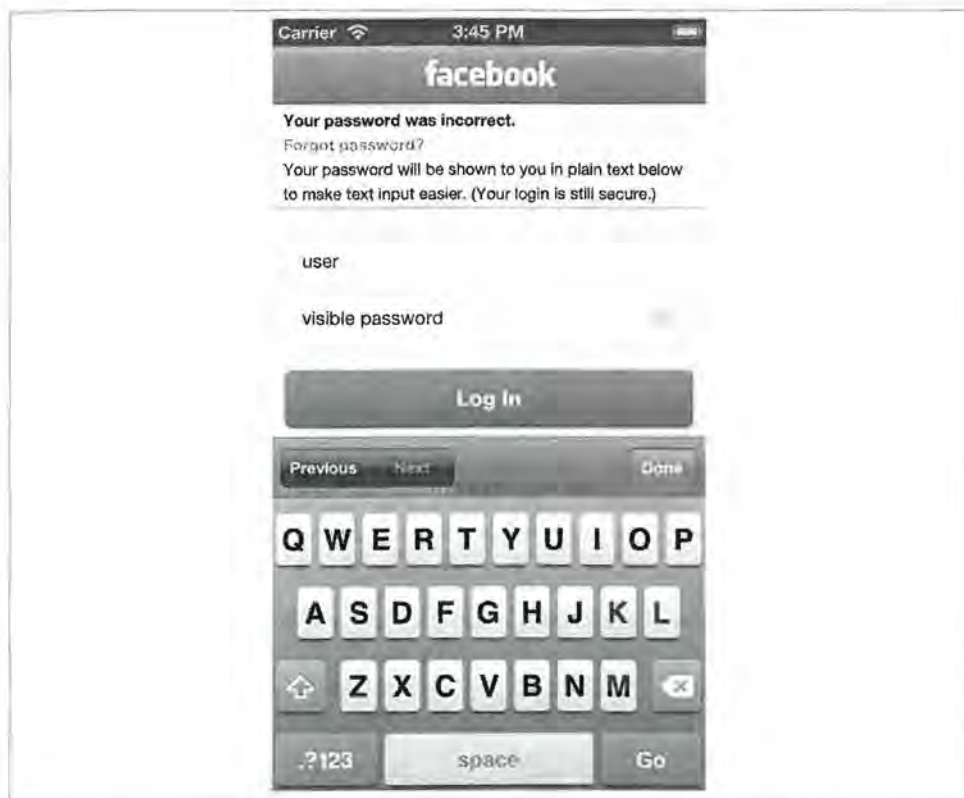


Figure 8-7. Facebook makes the password visible after a login error to improve usability.



For some older devices, such as BlackBerry 5.0 devices, we can also define that we want a numeric input box using the XHTML MP input `mode="digits"` attribute or using the WAP CSS `-wap-input-format` style.

Data lists for suggestions

HTML5 also adds a new `datalist` tag that is useful for autocomplete features. The data list is useful when an input type is open to any value, but we want to show the user some suggested values. As of this writing, the BlackBerry browser and IE10 for Windows 8 (not Windows Phone) support it.

We can define a data list with an ID and a set of child option elements:

```
<datalist id="dataCountries">
  <option value="France">
```

```

    <option value="Portugal">
    <option value="Spain">
  </datalist>

```

Then, we can use that list for suggestions in a text input, matching the `list` attribute with the data list's `id`:

```

<input type="text" id="txtCountry" list="dataCountries">

```

The data list will not have any UI if it is not associated with one or more form elements. When the user focuses on an associated text input and starts typing, the browser will suggest matching options from the data list.

Multiline text controls

When more space is required (like in a mail message body), we can use a `textarea`, and depending on the device the experience should be the same as with a normal text input. The biggest problem with multiline input controls is what happens when the content is larger than the available space. The scrolling mechanism inside a `textarea` element is a usability problem. To solve it, we can create an autogrowing text control.

This UI pattern was created by the Google Mobile team and is currently used in Gmail. The problem is that if we have a large amount of text in a `textarea`, scrolling inside it is very painful in some browsers (Safari on iOS is one of them). The solution is to grow the `textarea` to fit the contents, so the user can use the normal page scrolling instead of the `textarea`'s.



HTML5 supports several new attributes for `textareas`: `maxlength` to constrain the length of the content, `wrap="hard"` to add a newline character at the end of each line when sending the data to the server, `placeholder` to show placeholder text in the text area, and the new attributes that can be applied to every control, such as `autofocus` or `required`.

We can capture the `onkeyup` event and grow the `textarea` if necessary. We also need to capture `onchange` because pasting in iOS doesn't generate an `onkeyup` event.

The complete solution is available at www.mobilexweb.com/go/autogrowing. The code, borrowed from the Google Code Blog with a few changes, is:

```

<script>
// Value of the line-height CSS property for the textarea.
var TEXTAREA_LINE_HEIGHT = 13;

function grow(event) {
  var textarea = event.target;
  var newHeight = textarea.scrollHeight;
  var currentHeight = textarea.clientHeight;
  if (newHeight > currentHeight) {

```

```

        textarea.style.height = newHeight + 5 * TEXTAREA_LINE_HEIGHT +
        'px';
    }
}
</script>
<textarea onkeyup="grow(event);" onchange="grow(event);" >

```

Figure 8-8 shows the result.

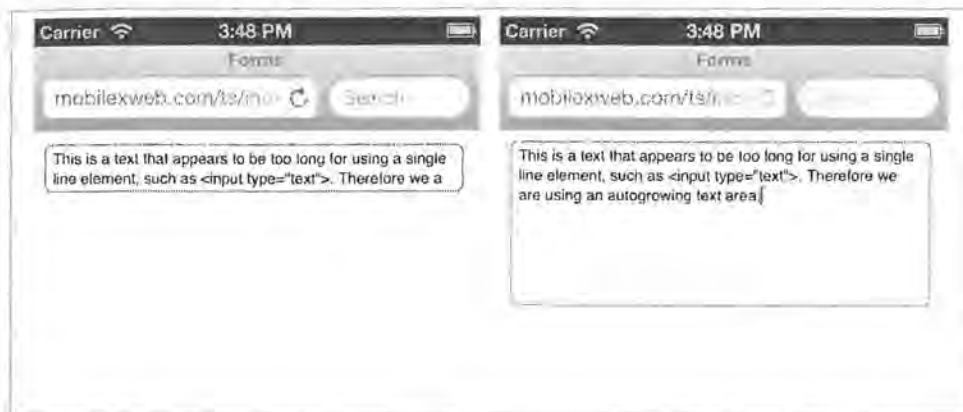


Figure 8-8. The autogrowing multiline textarea in action—the height of the control changes dynamically as the content grows.

Rich text controls

Rich text input controls are not very common in the mobile world, and they were really difficult to implement for many years because of the browser internals. Usually, desktop-based rich text input controls used iframes and were not compatible with mobile browsers.

On touch smartphones, the onscreen keyboard usually takes up too much space on the screen to have a toolbar around. However, on some larger devices, such as the iPhone 5, we can fix a toolbar over the keyboard zone and still have some space to type. On tablets we have enough space to provide a rich input control, and some solutions are appearing to provide this feature.



The BlackBerry 10 browser supports an extension HTML attribute, `data-blackberry-end-selection-on-touch="on"`, that disables the default behavior when the user selects text on the screen: after a selection is done, instead of opening the clipboard menu on tapping, it will just deselect the text.

contenteditable. HTML5 officially includes something that has been around on the Web unofficially for many years: the `contenteditable` attribute. This attribute accepts `true` or `false` as its possible values; most browsers will also parse it as a Boolean attribute, where its mere presence means true.

When we use `contenteditable="true"` on any HTML element, the whole element and its content becomes an editable control. As weird as it sounds, you can make a whole body element editable!

Safari on iOS from version 5 was the first mobile browser supporting `contenteditable`; when you tap on the editable element the keyboard and cursor will appear on the screen. In Figure 8-9, we can see this feature in action on the Windows Phone.



Figure 8-9. Here we can see `contenteditable` in action on Windows Phone with Internet Explorer—the whole section becomes a rich editor.

`contenteditable` compatibility as of the time of this writing is shown in Table 8-1.

Table 8-1. contenteditable compatibility

Browser/platform	Supports contenteditable
Safari on iOS	No until iOS 4.3 Yes from iOS 5.0 (contextual menu from 6.0)
Android browser	No until Android 3.2 Yes from Android 4.0
Chrome for Android	Yes
Internet Explorer	Yes
Nokia Browser for Symbian	No
Nokia Xpress browser for S40	No
Firefox	Yes
BlackBerry browser	No on smartphones up to 7.1 Yes on PlayBook from 2.0 Yes on BB10
webOS	Yes from 3.0 (tablet only)
NetFront	No
UC browser	No
Opera Mobile	No until 12.0 Yes from 12.1
Opera Mini	No

From iOS 6, Safari offers a new feature for content-editable elements: when the user selects text, a menu will appear offering the user the ability to apply bold, italic, or underline styling to that text, as we can see in Figure 8-10. This feature adds the proper HTML elements to the content itself.

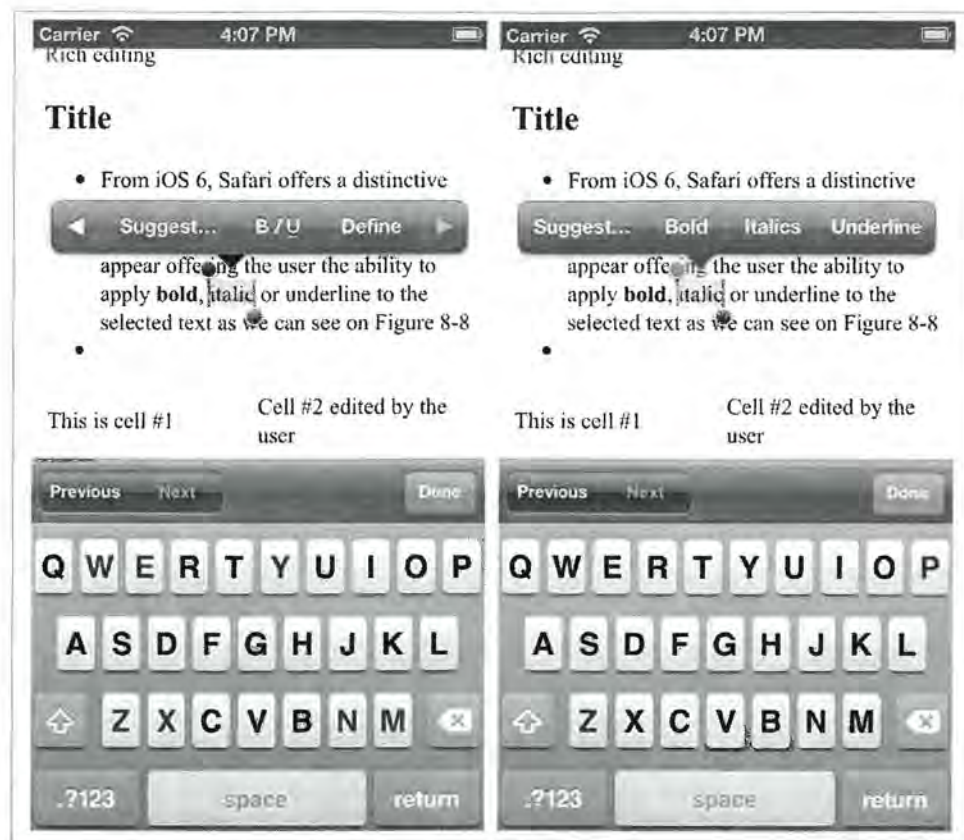


Figure 8-10. On iOS 6 and beyond, Safari shows a contextual menu on selections allowing you to apply bold, italic, and underline styles to them.



From iOS 6, Safari will always add a submenu for applying bold, italics, or underline styles to selected text. If you don't want this feature, be sure to remove those tags from the element after the user has ended the editing action.

If you want the user to be able to apply rich formatting—such as color, font size, bold/italics, or any other HTML styling—you can provide a toolbar near the editable content that will wrap the current selection in a new DOM element. When the user is typing new text or modifying current elements, she is changing the DOM directly, as if she were writing the `innerHTML` property of your content-editable element.



On compatible devices, it's a good idea to offer a fixed toolbar so it will not disappear when the user scrolls the content area.

I know what you're thinking: "How can we detect changes in the editable element?" The `onchange` event is not useful here because elements like a `div` or a `section` don't have that event, unlike form controls. There are four possible solutions:

- Read the `innerHTML` property of the desired element when the user submits the info, or frequently with a timer.
- Capture key events on the element, so we detect every keypress (or a specific keypress, such as the Return key).
- Use a *DOM mutation observer*, a modern technique available in some mobile browsers that will monitor any changes to the DOM, including the ones defined by the user on a `contenteditable` element.
- Use *DOM mutation event listeners*, such as `DOMNodeInserted`, `DOMNodeRemoved`, and `DOMCharacterDataModified`.

To provide our own rich toolbar options, we can use the `document.execCommand` method available in browsers that support `contenteditable`.

The execution command mechanism receives three arguments: the command's name, whether the default UI for the command should be shown, and some optional extra value information for the command. Commands include features such as apply bold, change colors, clipboard actions, and undo. Not every action is supported on each mobile browser, so we should test them before use.



All of these features were included in Internet Explorer for desktop, and now they are being discussed by the W3C as the HTML Editing API. We can find more information and a complete list of available actions at W3C's website (<http://www.w3.org/community/editing>).

WYSIHTML5. WYSIHTML5 is an open source rich text editor created fully in HTML5 that is compatible with mobile browsers. You can download it and see a live demo (<http://xing.github.com/wysihtml5>).

This framework will downgrade to a normal and basic `textarea` when the feature is not compatible with the current browser. The library supports a good way to define your own toolbar using just HTML5 elements, as in:

```
<a data-wysihtml5-command="italic">apply italic</a>
<a data-wysihtml5-command="foreColor" data-wysihtml5-command-value="red">
  apply red</a>
```

Every action can be applied to the current selection or, if there is no selection, the next values that the user types.

Pop-up controls

If we just need one quick text input, we can use the standard JavaScript prompt dialog instead of an HTML element. As you can see in Figure 8-11, the dialog has a nice native UI on different mobile browsers. Unfortunately, there is no way to customize the dialog's title or the virtual keyboard type.



Figure 8-11. Using the standard dialog prompt, we can replace single-element forms.

You use it as follows:

```
var input = window.prompt("Enter your details", "");
```



When we are creating native web experiences, the default JavaScript dialog is not good enough as it shows *file://* or the name of the HTML file as the dialog title. However, many native web platforms allow us to use a nonstandard native dialog instead.

Range Slider Fields

HTML5 includes the `<input type="range">` that will render on compatible devices as a range slider on the screen. The input needs at least the maximum (`max`) attribute defined so it knows the limit; you can also define a minimum (`min`) attribute different than the default zero value, and the `step` attribute, which will define the jump that the slider will make. For example:

```
<input type="range" min="0" max="100" step="10">
```

The range input type has some usability problems on mobile browsers:

- The draggable selector is too small on most mobile browsers, as you can see in Figure 8-12, and it's difficult to move it.
- The user does not see the current value in all browsers, which can make the control useless.



Internet Explorer 10 is one of the browsers that show the current value of the range input type while the user is dragging the control.

Because of these usability problems, the UI framework jQuery Mobile enhances the range input control with a much nicer user interface, including a bigger scrollable element and a numeric input that is synchronized with the slider automatically, as shown in Figure 8-12.

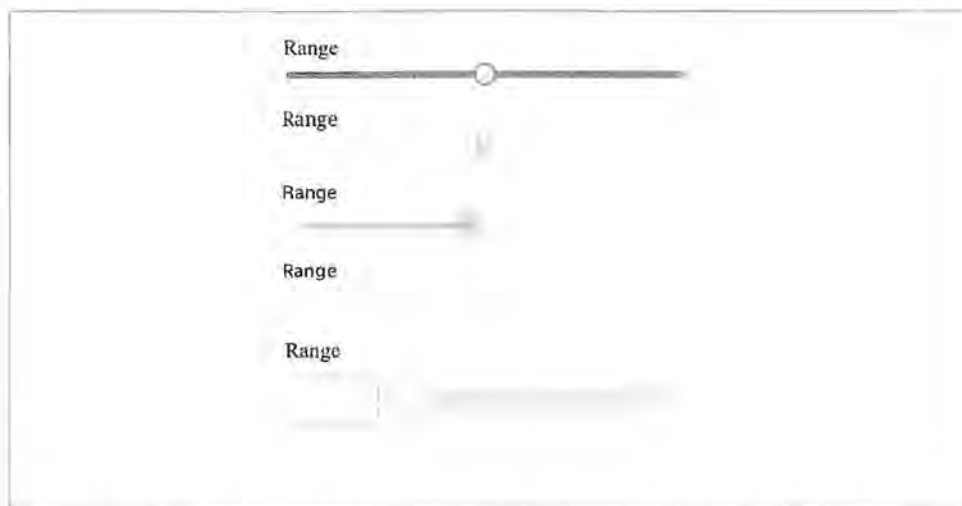


Figure 8-12. The standard `type=range` slider element in Safari on iOS, the Android browser, Google Chrome for Android, and the BlackBerry browser, and the jQuery Mobile enhanced control.

Date Input Fields

HTML5 defines new input types for date and time selectors. Unfortunately, not every mobile browser supports this feature at this time, but the good news is that if the browser is not compatible, the input field will fall back to a normal text input field.

The following type values are available:

- Date selector: `<input type="date">`
- Time selector: `<input type="time">`
- Date and time selector: `<input type="datetime">`
- Local date and time selector: `<input type="datetime-local">`
- Month selector: `<input type="month">`
- Week selector: `<input type="week">`



The date and datetime selectors will give you a string value in the value property. However, the browser guarantees that if you instantiate a new Date object with that string it will create a proper and valid date object.

Figure 8-13 shows how Safari on iOS and Chrome for Android render the different input types, usually as a pop-up control. The datetime selector will include the time zone when getting the value, but the datetime-local selector will not. The week selector is not compatible with most mobile browsers, so it will fall back to a text input control.

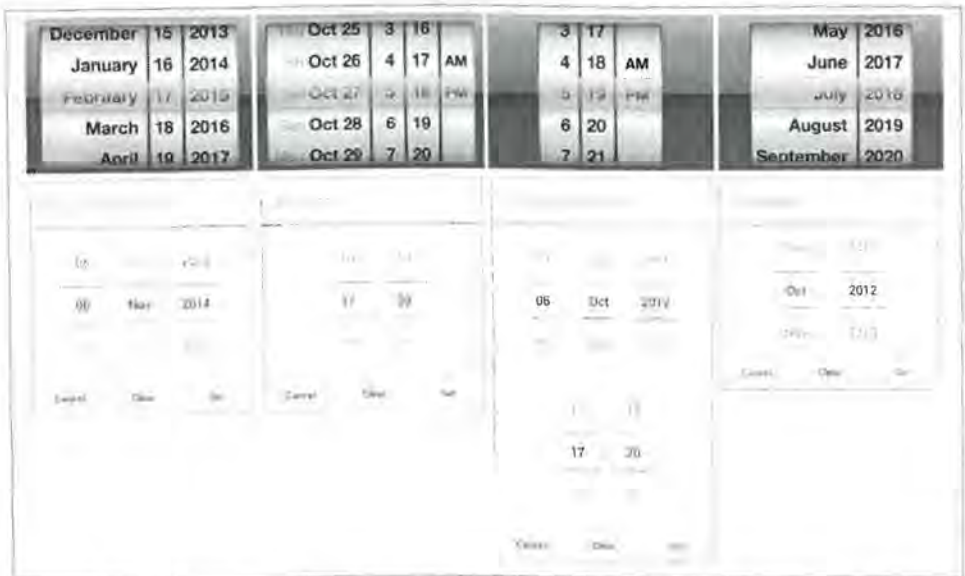


Figure 8-13. The date selectors in action (in order: date, datetime, time, and month—the datetime-local selector is the same as datetime, and the week selector is not compatible). Chrome for Android has removed support for datetime from version 26.



HTML5 defines a color picker control `<input type="color">`. However, at the time of this writing, only Opera for Android and BlackBerry 10 support it; all other browsers will fall back to a text input control.

File Selection Fields

The file selection control is the control typically used to browse for a local file to send to the server. When the user selects this control a modal pop-up window appears, allowing him to select a file from the public internal memory folders or from the additional memory card. A simple upload form might look like this:

```
<label for="photo">Upload a photo</label>
<input type="file" name="photo">
```

HTML5 adds two attributes to the `<input type="file">` element: the `multiple` Boolean attribute for multiple file selection and the `accept` attribute that allows us to filter the kind of files we want. We can choose to accept:

- Images only: `image/*`
- Videos only: `video/*`
- Audio only: `audio/*`
- Any specific format (MIME type)
- A comma-separated list of MIME types

Let's take a look at some samples:

```
<label for="manyfiles">Upload multiple files </label>
<input type="file" name="manyfiles" multiple>

<label for="video">Upload only a video</label>
<input type="file" name="video" accept="video/*">
```

Some mobile browsers, such as the Android browser up to 2.1 and Safari on iOS up to 5.1 inclusive, are not compatible with any kind of file upload. Therefore, if you use a file upload selector it will be rendered on the screen as disabled, as shown in Figure 8-14.

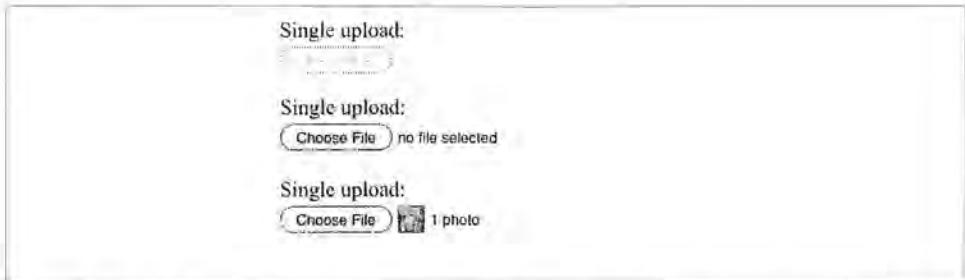


Figure 8-14. The file upload input type will not work on older devices, but here it's being rendered properly on newer ones.

Figure 8-15 shows a QR code that you can use in your own mobile browser to test for yourself how this new attribute works on your device.



Figure 8-15. If you want to test the file upload or any other code in this book by yourself, just point your mobile browser to <http://t.ad.ag> or scan this QR code.

Some browsers, such as Safari on iOS from version 6, Firefox for mobile, and Chrome for Android, will invite the user to use the camera when `accept="image/*"` is defined.

Actions with files

When the user selects one or more files, we can:

- Send the file to the server using POST and `enctype="multipart/formdata"` in the form element.
- Use the File API (to be covered in Chapter 15) to do something with the file locally using JavaScript. We can manage it as a binary file or, if it's an image, as a data URI element.
- Use XMLHttpRequest 2 (Ajax) to upload the file to the server and use progress events to show a progress bar while the file is being uploaded.

The HTML Media Capture extension

The W3C has released an HTML extension called *HTML Media Capture* (<http://www.w3.org/TR/html-media-capture>), which allows us to make use of the device's camera and microphone from an `<input type="file">` including the new capture Boolean attribute.



An old spec defined capture as a string attribute instead of a Boolean, supporting camera, camcorder, microphone, or filesystem as values. Version 2.x of the Android browser, Google Chrome up to version 18, and other browsers still support the old spec; if you use the Boolean version, they will just ignore it.

The `accept` attribute defines how to capture the media. Therefore, if we accept only images (`image/*`) or audio (`audio/*`) and we define a `capture` attribute, if the HTML Media Capture extension is available the user will get the camera or recorder app instead of a file picker when she clicks on the Browse icon (Figure 8-16).



Figure 8-16. With HTML Media Capture support, when the user clicks on “Browse” or “Choose File” the camera app or the sound recorder app will be opened directly as a picker for a temporary file.

Using the File API, we can get the file from the user and do some actions on the client side, such as resizing the image or performing face or QR code recognition.

With the new spec, the code should look like the following:

```
<label for="photo">Take a Picture now </label>
<input type="file" name="photo" accept="image/*" capture>
```

```

<label for="video">Record a video now</label>
<input type="file" name="video" accept="video/*" capture>

<label for="mic">Record audio now</label>
<input type="file" name="mic" accept="audio/*" capture>

```

For browsers still supporting the old spec, such as Android browser 2.x, the code must look like the following:

```

<label for="photo">Take a Picture now </label>
<input type="file" name="photo" accept="image/*" capture="camera">

<label for="video">Record a video now</label>
<input type="file" name="video" accept="video/*" capture="camcorder">

<label for="mic">Record audio now</label>
<input type="file" name="mic" accept="audio/*" capture="audio">

```

Usually, on mobile browsers HTML Media Capture will work only on nonmultiple file controls, as most camera applications do not have a specific UI to take a group of pictures and send all those pictures back to the browser at once. Therefore, if you want to use HTML Media Capture for multiple files you will have to either add several controls or add a new control dynamically with JavaScript after every selection.

Mobile browser compatibility

Safari on iOS was not compatible with file upload at all until iOS 6, nor was Android until 2.2; the webOS browser does not support file upload either. Therefore, there is no way to upload a file to a website on those devices. Remember that the original iPad and iPod touch (first generation) are not compatible with iOS 6; if you have users with these devices, they will see just a disabled element (as shown in Figure 8-14).

Safari on iOS from version 6 supports the `multiple` attribute and the `accept` attribute, but only with a value of `image/*` or `video/*`. Any other combination will just give the user a way to upload images or videos—there is no way to upload audio or other kinds of files, such as a PDF.

While Safari on iOS 6 does not support HTML Media Capture, if you use `accept="image/*"` or `accept="video/*"` without the `multiple` attribute, the user will be prompted to pick a file or open the camera application at that time to take a picture or record a video (see Figure 8-17).



Figure 8-17. Safari on iOS from version 6 will show different dialogs depending on the `accept` attribute's current value—if you add `multiple`, the gallery is shown by default and you can't use the camera app directly.

On Android devices, usually every browser (Chrome, the Android browser, Firefox, Opera) will ask for an *Intent* to pick a file based on the current `accept` value. Intents are a way on Android to integrate all the installed apps in an agnostic way—for example, if an app subscribes as an image picker, it will appear as an option when the user taps on a file upload control accepting only images, as you can see in Figure 8-18.

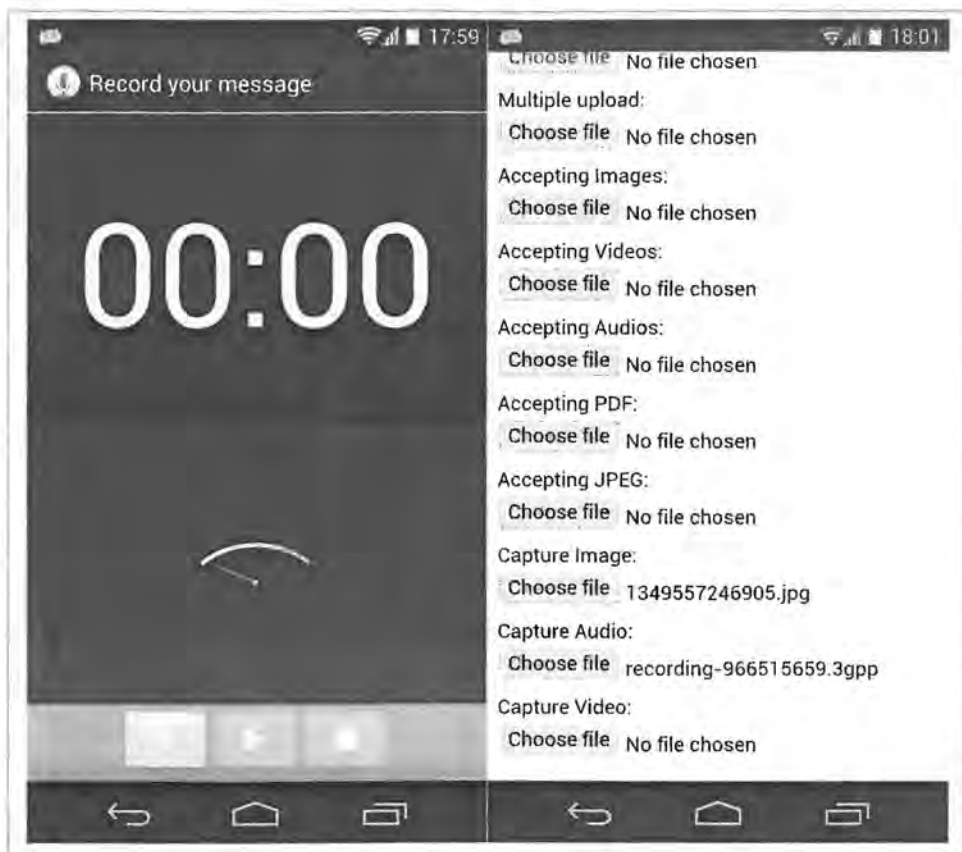


Figure 8-18. The Android browser and Google Chrome for Android show an Intent selector where you can choose which installed native app to use to pick the file, based on the accept attribute's current value.

As you'll see in Table 8-2, some browsers on Android devices support HTML Media Capture; if the capture attribute is included, the proper app (camera or voice recorder) will be opened automatically when the control is activated without any dialog being shown.

Browsers such as Nokia's Xpress browser for S40 and Opera Mini running on social phones will open a filesystem browser that will look into the internal or external memory to select a file and upload it every time they encounter a file upload control.

We can classify mobile browsers in terms of file upload compatibility as having:

No compatibility at all

Your field will be disabled.

Basic compatibility

Users can upload a single file, but there is no filtering ability and no HTML Media Capture support. Some browsers will offer users the ability to capture the file on the screen at that moment using the camera app.

Filter compatibility

Users can upload only a single file, but you can use the `accept` attribute to filter files (usually at least images and videos).

HTML5 compatibility

Users can upload one or more files, and the `accept` attribute works fully.

HTML Media Capture compatibility

All possible options are available. When using capture, usually multiple can't be used at the same time.

Table 8-2 summarizes browser compatibility at the time of this writing.

Table 8-2. File upload compatibility table

Browser/platform	Basic support	File picker/capture	accept attribute	multiple attribute	capture attribute
Safari on iOS	No <= 5.1 Yes, 6.0+	Picker and camera capture	Yes, only images and videos	Yes	No
Android browser	No <= 2.1 Yes, 2.2+	Picker, camera, and voice capture	Yes, only images, videos, and audio	No	No <= 2.3 Yes, 3.0+
Chrome for Android	Yes	Picker, camera, and voice capture	Yes	No	Yes
Internet Explorer	No <= IE9 Yes, IE10+	Picker and camera capture	Yes	Yes	No
Nokia Browser for Symbian	Yes	Picker	Partial (shows warning)	No	No
Nokia Xpress browser for S40	Yes	Picker	No	No	No
Firefox	Yes	Picker, camera, and voice capture	No	No	No
BlackBerry browser	Yes, 4.2+	Picker	No	No	Yes on BB10
UC browser	Yes	Picker and camera capture	No	No	No
Opera Mobile	Yes	Only picker	No	No	No
Opera Mini	Yes	Only picker	No	No	No

Noninteractive Form Elements

In addition to supporting labels and hidden fields (`<input type="hidden">`), HTML5 defines some new noninteractive elements that are related to forms or value definitions. The new elements are `output`, `progress`, and `meter`. While not every browser renders these new elements properly at this time, we always have a fallback mechanism.

The `output` element is intended for the result of a calculation on a form. It receives a `for` attribute whose value is a space-separated list of controls involved in the calculation and a `form` attribute if we are using it outside of a `form` element.



You can make any interactive form control read-only by applying the new HTML5 Boolean attribute `readonly`.

For example, if we were working on a currency calculator project we might have a form like the following:

```
<form id="currencyCalculator">
  <label for="value">Value in US Dollars</label>
  <input type="number" id="value">

  <label for="currency">
    <select id="currency">
      <option value="EUR">Euro</option>
      <option value="GBP">British pound</option>
      <option value="JPY">Yen</option>
    </select>

    <input type="button" value="Convert">
  </form>

  <output for="value currency" form="currencyCalculator"></output>
```

We could use JavaScript, Ajax, or a server postback to fill the output value.

The `progress` element provides a way to render a progress bar on the screen:

```
<progress>Loading...</progress>
```

When the browser is not compatible, it will just show the element's contents (here, "Loading..."). An indeterminate progress element (with no `value` attribute) is used when we don't know how much time (or what percentage) is left to complete the operation. Usually, a browser will render it as a progress animation—for example, Internet Explorer 10 shows a points floating animation typical of the Windows Phone and Windows 8 user interface.



If you are creating Windows 8 Store apps with HTML5—not browser-based apps—Microsoft allows a progress bar to be rendered as an animated ring using the nonstandard style `-ms-progress-appearance: ring`.

If we know the current value and the maximum value, we can create a determinate progress bar using the `value` and `max` attributes, which can be changed dynamically using JavaScript:

```
<progress max="100" value="60">60%</progress>
```



If you don't provide a maximum value for a progress bar, a floating-point value between 0 and 1 is expected.

If we want to show a fractional value that is not expected to change as a progress indicator, we can use the `meter` element. For example, to show the available space on some account:

```
<meter max="1000000" value="450000">
```

`meter` is not yet well implemented in mobile browsers, but it supports more semantic attributes to indicate how good the current value is—we can specify `optimum`, `low`, and `high` values that may be used in the future by compatible browsers to show different areas on the meter in different colors.

Figure 8-19 shows the `progress` and `meter` elements rendered in two browsers that support these new elements.

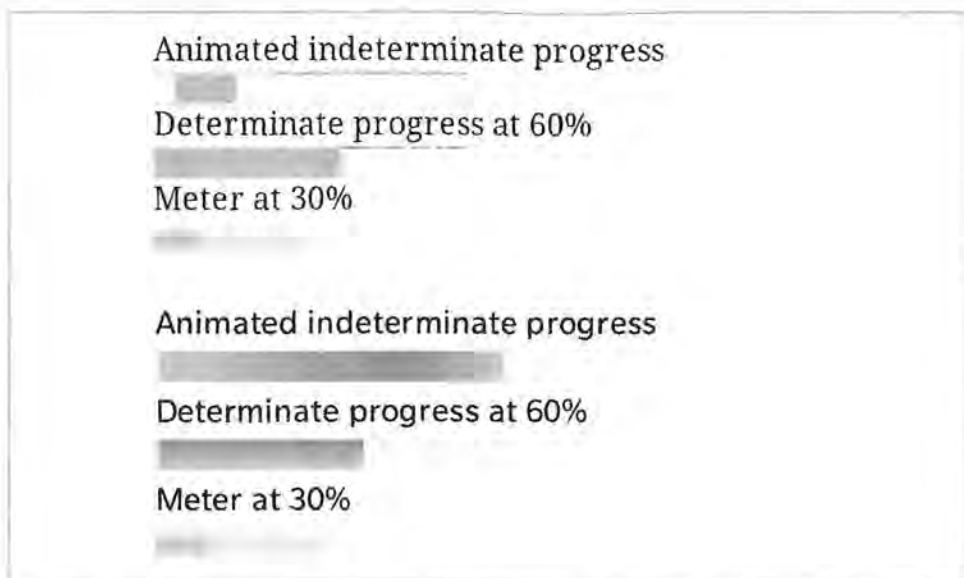


Figure 8-19. The progress and meter elements rendered in Google Chrome for Android and the BlackBerry 10 browser.



HTML5 also includes the `keygen` element, which generates a public/private key pair and sends the public key to the server in the form. Mobile usage at the time of this writing is quite rare and compatibility is not yet good enough for me to recommend its use.

Form Control Attributes

HTML5 includes new attributes that can be applied to almost every form element, including `input`, `textarea`, and `select`. Some of the attributes are semantic flags, and some others provide new behavior.

Placeholder

A placeholder is a hint that is shown inside a text box until the user inserts text in that field. When the user starts typing, the placeholder is hidden, as shown in Figure 8-20. This feature is very useful in mobile designs because of the lack of space to add more information or hints about what the field should contain.

URL

http://www.example.com

Figure 8-20. The placeholder is the gray descriptive text inside the text box that is automatically deleted when the user begins typing a value in the box.

The placeholder attribute is available in HTML5, and for noncompatible browsers we can create a little script to give this functionality, even using the standard placeholder attribute as the source:

```
<input type="text" name="zip" placeholder="Your ZIP Code">
```

We can customize the appearance of the placeholder using some nonstandard CSS hacks, such as `::-webkit-input-placeholder` for WebKit-based browsers, `:-ms-input-placeholder` for Internet Explorer 10, and `:moz-placeholder` for Firefox:

```
input::-webkit-input-placeholder, input:-ms-input-placeholder,  
input:moz-placeholder {  
    color: #AFEFAF;  
}
```

autofocus

Adding the autofocus Boolean attribute to a control tells the browser to focus on that element once the page is loaded, so we don't need to rely on JavaScript hacks to support that feature. For example, if we are working on a search engine, we can focus on the main search field using:

```
<input type="search" autofocus>
```



A JavaScript library can add support for placeholder and autofocus even on devices with no support for those attributes. You can download this library from GitHub (<http://gist.github.com/330318>).

autocomplete

The autocomplete attribute accepts an on or off value, and it can be applied to the form element as a whole or to individual input elements. This allows us to enable or disable the autocompletion or autosuggestion features that the browser may use in forms.

readonly

If for some reason we don't want the user to interact with a form element, we can disable it with `disabled` or make it `readonly`. Both attributes are Boolean:

```
<select disabled></select>  
<input readonly type="tel">
```

Input Validation Attributes

As we'll see later in this chapter, HTML5 includes a validation feature that can be used by JavaScript or CSS. In this category, we can find the following new attributes:

`required`

A Boolean attribute that can be applied to any input type

`min`

Specifies the minimum valid value; can be applied to numeric (number or range) or date input types

`max`

Specifies the maximum valid value; can be applied to numeric (number or range) or date input types

`pattern`

Specifies a regular expression against which the input value should be validated

`step`

Used for numeric (number or range) input types; specifies the legal number intervals

Safari Extensions

Safari on iOS includes `autocorrect="{on|off}"` and `autocapitalize="{on|off}"` attributes, to activate/deactivate automatic spelling correction (preferred for non-dictionary input fields) and automatic capitalization for the input.



For BlackBerry 5.0 devices, we can also define that we don't want prediction on our input fields using the XHTML MP `inputmode="predictOff"`.

Firefox Extensions

Firefox for mobile devices includes the `mozactionhit` attribute that is used to define the label of the label/return key on virtual keyboards. The possible options are `go`, `done`, `next`, `search`, and `send`, and the value will be mapped to the appropriate string in the current language.

For validation purposes, Firefox shows an error bubble when a field doesn't comply with its constraints (for example, the email address does not have valid syntax). To replace the default error message, Firefox supports a nonstandard attribute, `x-moz-errormessage`:

```
<input type="text" name="name" required x-moz-errormessage=
    "You must enter your name">
```

XHTML Mobile inputmode

Some non-HTML5 devices, such as BlackBerry smartphones running OS 5.0, also accept the XHTML Basic/Mobile Profile `inputmode` attribute with a comma-separated list of tokens. The tokens can be script tokens (language charsets) or modifier tokens.



The `inputmode` attribute is also defined by WHATWG in the HTML specification, but not by the W3C.

The available script tokens are:

- arabic
- bopomofo
- cyrillic
- georgian
- greek
- han
- hangul
- hebrew
- hiragana
- kanji
- katakana
- latin
- simplifiedhanzi
- thai
- traditionalhanzi
- user

The possible modifiers are:

- `lowerCase`
- `upperCase`
- `titleCase`
- `startUpper`
- `digits`
- `symbols`
- `predictOn`
- `predictOff`

Additional Form Attributes

Before HTML5, every form control needed to be inside the `form` element to work without JavaScript. From HTML5, there are new attributes that we can apply to form elements that define to which form a control should be applied. We need to be careful on mobile browsers, though, as compatibility is not very good yet and there is no simple way to create a fallback.

In this category, we can find attributes such as `form` (to specify the IDs of one or more forms that an `input` element belongs to), `formnovalidate` (for elements that should not be validated on form submission), and button-specific attributes such as `formaction`, `formenctype`, `formmethod`, and `formtarget`.

Form Validation

To reduce the number of client-side scripts and server-side trips required for validation and to improve the usability of our forms, we should provide as many input validation properties as we can.

The first typical option is to define the maximum size accepted for the text input using the `maxlength` property, expressed as a number of characters. Many platforms automatically add a character counter while the user is typing. A second popular option is to define the right input type for `input` elements, such as `number` or `date`. That will definitely improve usability and validation.

To add more layers of validation without our own custom algorithms, HTML5 and WAP CSS (from XHTML MP) offer us some solutions.

HTML5 Validation

As we saw earlier, HTML5 adds some validation attributes, such as `required`, `pattern`, `min`, and `max`. The question is, what happens if a required input field is left without content, or if a numeric input field has a value greater than the `max` definition? What happens if the user doesn't type a valid email address in an `<input type="email">` field?



Even though not all browsers support the HTML5 validation attributes, it's a good idea to use them as flags for our JavaScript validation algorithms instead of hardcoding the validation rules into our scripts.

The behavior changes per mobile browser, but here are some of the things that may happen:

- A validation bubble appears if the control doesn't validate against the rules.
- The form can't be submitted until the whole form validates against the rules.
- The JavaScript API is used to validate the input.
- We can use the new CSS validation pseudoclasses to show the validation process dynamically.

On desktops, most browsers add a validation bubble or halt the submission if validation rules are not met. However, most mobile browsers ignore that and leave it up to our CSS or JavaScript code to enforce the validation rules.

At the time of this writing only Firefox halts submission and shows an error bubble by default, as we can see in Figure 8-21.



Figure 8-21. Firefox shows an error bubble when the element is not valid—we can define the error text using the nonstandard `x-moz-errormessage` attribute.



If you want to disable validation in some situations, you can use `<form novalidate>` or apply the `formnovalidate` attribute to one specific button, as in `<input type="submit" formnovalidate>`.

CSS validation pseudoclasses

CSS3 added new validation pseudoclasses that can be applied dynamically to form elements. The available pseudoclasses are:

`:required` and `:optional`

Applied automatically for elements with or without the `required` Boolean attribute

`:read-only` and `:read-write`

Applied automatically for elements with or without the `readonly` Boolean attribute

`:valid` and `:invalid`

Applied automatically for all the rules, such as `type`, `min`, `max`, `pattern`, and `required`

`:in-range` and `:out-of-range`

Applied automatically when the `min` and `max` attributes are defined



If you don't use `required` on an attribute but do use another rule such as `pattern`, `type`, or `max`, an input without content will always be validated as valid. If the user enters at least one character, the other rules will define the validity of the input.

With these new pseudoclasses, we can change borders or backgrounds for inputs when they become valid or invalid:

```
input:valid {  
    border: 1px solid green;  
}  
input:invalid {  
    border: 1px solid red;  
}
```

We can also mix these new features with other pseudoclasses such as `:focus`, so we can define a different style for invalid input types only when they are in focus:

```
input:focus:invalid {  
    border: 1px solid red;  
}
```



WebKit-based browsers that show a bubble error message on submit allow customization through pseudoelements. You can customize styles using `::-webkit-validation-bubble`, `::-webkit-validation-bubble-message`, `::-webkit-validation-bubble-arrow`, and `::-webkit-validation-bubble-arrow-clipper`.

Constraints validation API

HTML5 comes with a new API for form constraints validation. On compatible devices, the `form` element has a `checkValidity()` method that runs the constraints validation evaluation process and returns a Boolean value indicating the result, and a `setCustomValidity(message)` method that allows us to force a false validation result because of a custom validation process. We can use this method inside a `change` or `input` event listener.

Every form element, such as an input element, also has the following properties:

`willValidate`

A Boolean value that defines the current state of the constraints evaluation. A false value means there is a problem.

`validity`

A `ValidityState` object defining Boolean attributes for every possible constraint problem, such as `validity.typeMismatch`, `validity.valueMissing`, `validity.customError`, `validity.patternMismatch`, `validity.rangeOverflow`, or `validity.rangeUnderflow`.

`validationMessage`

A message describing failures.



The `onchange` DOM event fires after the form element has lost focus. The new `oninput` DOM event available in some browsers will be fired any time the element is changed because of the user's intervention, such as pressing a key.

If we want to halt form submission if there is a problem in the validation process, we can use the following code:

```
<form onsubmit="return this.checkValidity()">
```

If we want to know whether an element such as `<input type="number" max="110" id="age">` has any problem with validation, we can use:

```
var ageInput = document.getElementById("age");
var isNotNumeric = ageInput.validity.typeMismatch;
var isOverflow = ageInput.validity.rangeOverflow;
```

When working with numeric input fields, we need to remember that the `value` property always returns a string version of the value. We can force a number using `parseInt` or `parseFloat`; modern browsers also support the new `valueAsNumber` attribute on input elements.



The constraints validation API doesn't remove the need for server-side validation scripts to avoid security issues.

WAP CSS Validation

As mentioned earlier, WAP CSS added the property `-wap-input-format`, which allows us to define the type and number of characters that the user can input (known as the *input mask*). Specifying an input mask will reduce the user's error possibilities; it can yield error messages like the one shown in Figure 8-22. There is also the `-wap-input-required` attribute, which prevents the user from moving the focus away from a field until she has entered some text in it.



Figure 8-22. Browsers compatible with WAP CSS input constraints show some kind of error message when the user enters text in an invalid format and then tries to move the focus away from that field.

The content of the `-wap-input-format` attribute is a string mask using the special characters in Table 8-3.

Table 8-3. WAP CSS input format patterns

Pattern	Usage
a	Any character, letter, number, or symbol.
A	Any uppercase alphanumeric character.
n	Any numeric character or symbol.
N	Any numeric character.
x	Any lowercase alphanumeric character or symbol.
X	Any uppercase alphanumeric character or symbol.
m	Any character, lowercase by default, but with uppercase possible.
M	Any character, uppercase by default, but with lowercase possible.
{n}{pattern}	A fixed number of repeats (n) of the pattern defined. For example, 4N means four numeric-only characters.
*{pattern}	Any number of repeats of the pattern defined. For example, *A means any number of uppercase alphanumeric characters. It can be used only once per pattern.
{pattern}	Pattern combination. For example, A*a means one uppercase character and then any number of any other character.

We can also escape other characters to create complex patterns, but this is not recommended because the pattern matching engines are not the same on all platforms and strange behavior can result.



Internet Explorer Mobile before Windows Phone also accepted the nonstandard attribute `emptyok`, defining whether a value was required (false) or not (true).

The next sample shows the standard way to define a required US zip code text input, an optional phone number text input, and a required password numeric field using XHTML MP and WAP CSS:

```
<dl>
  <dt><label for="zip">ZIP Code</label></dt>
  <dl><input type="text" name="zip" style="-wap-input-format: '5N';
    -wap-input-required: true" /></dl>
  <dt><label for="phone">Phone Number</label></dt>
  <dl><input type="text" name="phone" maxlength="15"
    style="-wap-input-format: '*n';" /></dl>
  <dt><label for="password">Password</label></dt>
  <dl><input type="password" maxlength="8" name="password"
    style="-wap-input-format: '8N';
    -wap-input-required: true" /></dl>
</dl>
```

Many older XHTML mobile browsers understand the `inputformat` or `format` attribute, imported from WML. The syntax is the same as for the WAP CSS attribute. If we want to add the same feature for older Openwave-based devices, for example, we can use the `inputformat` attribute. Internet Explorer uses the `format` attribute with the same pattern inside:

```
<input type="text" name="zip" inputformat="SN" format="SN"
      maxchars="5" />
```


Feature and Device Detection

Before moving on with more HTML5, CSS, and JavaScript code that we can use on mobile devices, we need to pause and talk about feature and device detection. As we saw in the last chapter, not every browser supports exactly the same features, and even different versions of the same browser may have differences in terms of what we can use.

This book was likely finished a few months (or maybe even years) before you picked it up. I can assure you that what some browsers support today may differ from what they supported at the time I was writing it.

This is not just a book authoring issue. It's also a development issue, because we can't constantly update every website and web app we create to suit every new browser version or platform on the market.

Therefore, we need a way to design and develop our code that makes it as future-proof as possible. To that end, we are going to look at some techniques that will be useful for feature detection, device detection, and fallback mechanisms.

In combination with all these techniques, we can use the progressive enhancement technique (covered in Chapter 5) and some graceful degradation hacks to ensure that our code works as well as possible on as many platforms as possible.

Possible Problems

To find the right solutions, first we need to recognize all the possible problems we may need to deal with. We can summarize the possible problems with regard to features as follows:

- An HTML feature we want to use is not available, such as the progress element.
- A CSS feature we want to use is not available, such as CSS animations.

- A JavaScript API we want to use is not available or only partially available, such as Geolocation.
- We need to deal with different contexts: different input modes, screen resolutions, pixel densities, and screen sizes.
- A feature may not work properly even when it is officially implemented.
- Some APIs (such as video-playing codecs) are technology-agnostic, so we need to know current compatibility.
- We need to deliver content that is platform-specific, such as an Android native application, an ebook, or a Passbook coupon for iOS.
- We want to use a feature that is prefixed—that is, for which every browser has a different prefix, usually on CSS or JavaScript API.

Possible Solutions

Different types of solutions are available to solve problems we detect. For example, we can:

- Use internal HTML, CSS, or JavaScript mechanisms to provide a fallback, so if A is not available, B will automatically be executed.
- Use a *polyfill*, so if A is not available, some code is being executed on top of the browser that emulates A compatibility.
- Use a client-side feature detection framework, so if A is not available the framework will warn us somehow so we can use another solution.
- Use a client-side device detection mechanism, so if we already know that browser X doesn't support A, we can use another solution.
- Use a server-side feature library, so if A is not available, we change the code from the server or redirect the user to another page.
- Use a server-side device detection mechanism, so if we already know that browser X doesn't support A, we can change the code from the server or redirect the user to another page.

Which solution is the best one? Well, unfortunately there is no unique answer. Let's analyze it.

Device detection mechanisms should be avoided as much as possible, because as we know from Chapter 1 to Chapter 3 there are too many platforms and too many versions out there, and creating an infinite if-else-if algorithm (if it's iOS 6 then A, if it's iOS 5 then B, if it's Android then C...) is not a good idea for our mental health. However, in some specific situations, there is no other way to solve a problem than asking for the specific platform.



Device detection always creates new problems for the future. For example, until 2012 most websites were not delivering `<input type="file">` for iOS because it was not supported until iOS 6. So every developer must go now and change the condition if they realize that now iOS 6 supports them.

Generally, you should always try to rely on internal HTML, CSS, and JavaScript mechanisms to provide fallbacks, as we'll cover in the next few pages. However, you can't solve every problem this way, so using feature detection is the next recommended pattern. Client-side frameworks use automatic tests to mark a feature as compatible, but occasionally that may lead to false positives, or you may encounter features that cannot be detected automatically. Server-side feature libraries are manually tested server-side databases, so their reliability in some situations can be higher than that of automatic tests.

About polyfills

While a polyfill may seem like a good idea for classic web development in some situations, when developing for the mobile web I recommend avoiding them as much as possible. A polyfill is usually a JavaScript framework that fills a gap for some feature that the browser doesn't recognize natively: it offers compatibility by emulating the same behavior. (Of course, a polyfill for every feature is not always possible.)



HTML5 Please (<http://html5please.com>) is a community-updated website where you can find information on every HTML5 feature and advice on whether you can use it with confidence or should use it with a fallback, use a polyfill, or avoid usage.

The problem on mobile browsers is performance. Like every JavaScript framework, a polyfill will take time and battery power to “fill the gap,” and if a browser does not support a feature it may be because it's an old platform—and old platforms mean less memory and slower performance. Therefore, doing something that harms performance may be a very bad idea. That's not always the case, but you should at least test what the possible performance impact may be before deciding to use a polyfill.

Informational Websites

If we want to know the current state of one API on different platforms, we can use some informational websites that will help us to understand compatibility. In Chapter 6, we covered browser testing tools such as HTML5 Test (<http://html5test.com>) and Ringmark (<http://rng.io>) that will assess current compatibility dynamically. There are also

informational websites that provide compatibility tables to help us better understand current compatibility on all the browsers after manual or automatic tests.

Can I Use

The most well-known website that provides compatibility information for web browsers is Can I Use... (<http://caniuse.com>). It has a good amount of information on features and sub-features of CSS, HTML5, SVG, JavaScript APIs, and others. Every feature is compared across many desktop and some mobile platforms. On the mobile side, compatibility information is currently limited to Safari on iOS, Opera Mini, the Android browser, and the BlackBerry browser. One of the important features is that we can see every feature's compatibility in the latest version, previous versions, and expected future versions of every browser platform. The site also includes links to related articles for almost every feature that is listed.



If you want to run the tests that define the information on the Can I Use website, you can point your browser to Can I Use... test page (<http://tests.caniuse.com>).

Global usage share statistics from StatCounter Global Stats (<http://gw.statcounter.com>) are also provided for each feature, so we have an idea of the market share of compatible browsers. Unfortunately, at the time of this writing that data is not separated between mobile and desktop platforms, so the statistics are difficult to interpret from a mobile-only point of view.



If you want to see an overview graph of the Can I Use data, you can point your browser to the HTML5 and CSS3 Readiness website (<http://html5readiness.com>).

In Figure 9-1, we can see compatibility for the WebGL JavaScript API at the time of this writing.

MobileHTML5.org

Because I couldn't find any good and reliable resource on the Web specifically about HTML5 compatibility on mobile devices, I decided to create the website Mobile HTML5 (<http://mobilehtml5.org>), with the goal of covering as many mobile platforms as possible with real manual testing examples. Thanks to the fact that I own 45 mobile devices, I have the ability to manually test every feature on every platform; the results are displayed on the site in a quick-reference table to help viewers understand where we are in terms of compatibility (see Figure 9-2).

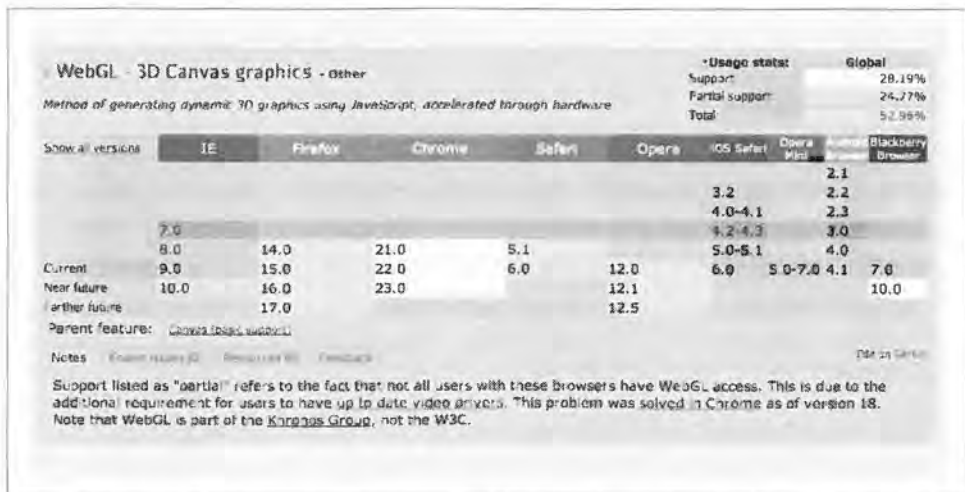


Figure 9-1. With Can I Use, we can check the current and future compatibility of dozens of features and abilities.

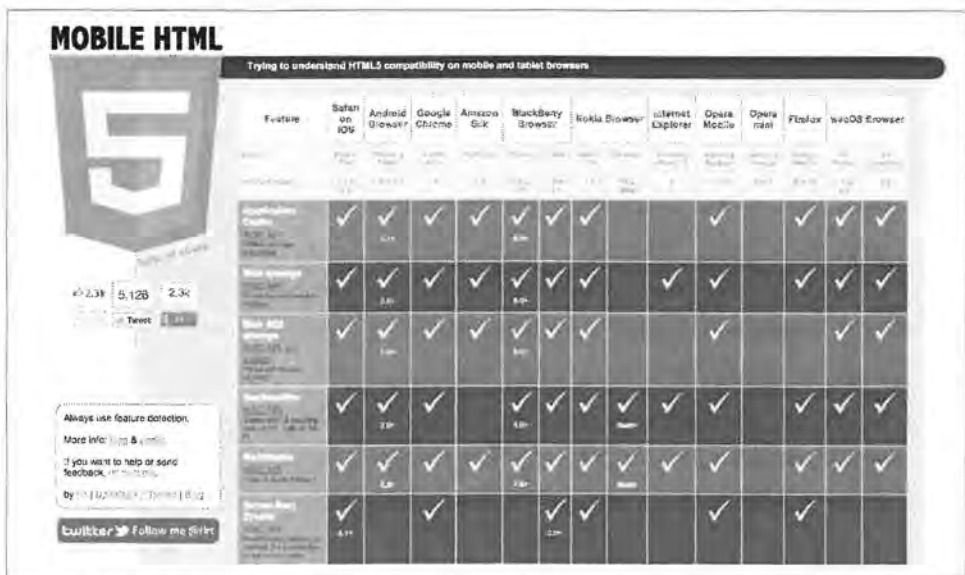


Figure 9-2. Mobile HTML5 provides a compatibility table containing up-to-date information on every mobile platform with at least some basic HTML5 support.

Most browser vendors are also helping me to maintain the website and ensure that the information published is correct and up-to-date.

The website's focus is on HTML5 JavaScript APIs and some big CSS features, such as animations.

WebPlatform.org

Working with a selection of big vendors such as Google, Opera, Nokia, Adobe, Apple, Microsoft, and Mozilla, the W3C started a project in late 2012 to document the Web and provide a central place where developers can find useful articles and compatibility information on HTML, JavaScript, and CSS topics.

It's a community project, so anyone can participate and contribute to it; as it grows, it will become a great resource to find articles, API documentation, and compatibility information.

To access it, just browse to Web Platform's website (<http://webplatform.org>).

Client-Side Detection

On the client side (in the browser), we can detect features, provide fallbacks, and issue queries about the current platform using a mix of technologies. Let's review them.

HTML, from its origins, has included a beautiful fallback mechanism that has helped the Web to grow without big incompatibilities. Basically, every browser will make the following assumptions without firing an error or stopping the rendering:

- If an element is unknown, the browser will ignore the open element, such as `<progress>` with all its attributes, and its closing element, such as `</progress>`; however, it will render its child content, such as other HTML elements or plain text.
- If an element is known but one or more of its attributes are unknown, the element will be rendered properly and the unknown attributes will just be ignored.
- If a CSS selector or CSS at-rule is unknown, the whole declaration will be ignored.
- If a CSS selector is valid but a style is unknown, only that definition will be ignored.
- If a CSS selector and a style are valid, but the value for the style is unknown, only that definition will be ignored.
- If the same CSS style on the same element is declared more than once, the last valid declaration will be used.

HTML Fallbacks

Providing fallbacks for new HTML elements is really easy. Hopefully, browsers will provide the fallback mechanism by default. Let's continue with the progress example:

```
<progress max="100" value="8">
  <img src='loading.gif'>
  Progress 8%
</progress>
```

Browsers supporting the progress element will completely ignore all the child elements: that is, the `img` element and the `Progress 8%` text. Browsers that don't support the progress element will just ignore it, and they will parse the `img` and the text properly.

When it comes to attributes, it's even easier: we just use any attribute we want, and noncompatible browsers will ignore it or take a default action. Therefore, `<input required>` will render properly on all browsers (with some ignoring the `required`), and `<input type="date">` will fall back to a `type="text"` input field on browsers that do not support date input controls.

CSS Fallbacks

Let's focus on CSS style declarations. If a browser doesn't understand a style it will just ignore it, without affecting any other declarations. For example:

```
#logo {
  background-image: url('logo.png');
  background-size: 50%;
}
```

The `background-size` style is new in CSS3; noncompatible browsers will just ignore it and use only the `background-image` style.

When we want to use a style value that may not be compatible with all browsers, we should use a multiple declaration, with the first value being the most widely compatible and the last one the most specific. Let's take a look at a sample:

```
#logo {
  background-image: url("logo.png"); // most compatible version first
  background-image: image-set(url("logo2x.png") 2x,
    url("logo.png") 1x));
}
```

In this example we want to use the new `image-set()` function that can be used to define different images for different resolutions. The problem is that it is not widely supported today, and if we use it on its own, noncompatible browsers will just ignore the whole declaration, meaning there will be no image for the logo.

The solution involves using two declarations for the same style. The first one should be the most compatible solution, which can be used by every browser. Browsers compatible with `image-set()` will also understand the last declaration, which will replace the first one. Noncompatible browsers will ignore the last declaration, considering it invalid, and the first one will win.

CSS conditionals

A recent W3C draft standard, CSS3 conditionals (<http://www.w3.org/TR/css3-conditional>) allows us to define @supports rules for conditional processing—that is, we can define new styles and provide alternatives for browsers that do not yet support them:

```
@supports (box-shadow: 2px silver) {  
  #element {  
    box-shadow: 2px silver;  
  }  
}  
/* Provides alternative when it is not supported */  
@supports not (box-shadow: 2px) {  
  #element {  
    border: 2px solid red;  
  }  
}
```

Unfortunately, at the time of this writing, there is no mobile browser supporting this ability.



Don't confuse CSS conditionals with CSS media queries. Conditionals query on browser CSS support, and media queries query on device or browser abilities.

Vendor Prefixes

According to the CSS standard, when a browser is implementing a feature that is experimental, proprietary, or not yet mature enough for general usage, or if the discussion of how to implement it is not yet finished, the implementation must use a prefix so developers know that the final standard may change.

That was a good idea when browsers and the Web were evolving slowly, over the course of a few years. However, today's browsers (and mobile devices) can change a lot in just a few months, and because of their nature, they are hungry for more and more features. Therefore, today we use more prefixed features than we really want to, and the prefix system seems somewhat obsolete and nonsensical.



The problem with prefixes is that we need to clone our code to define every possible prefix available today, such as `-webkit-` for WebKit-based browsers, `-moz-` for Firefox, `-o-` for Opera (Presto), or `-ms-` for Internet Explorer. Also, some values are not compatible between browsers, so we need to use different ones for each prefix.

There is a big discussion in the community about what to do with prefixes and the rules because we developers have seen that they make our website code longer, and harder to

read and maintain. I will keep that discussion out of this book, and focus just on describing the problem and some possible solutions.

When a feature is mature in terms of the standard discussion, browsers can stop requiring the prefixes and use the standard style directly. Let's look at an example. To make rounded corners, the CSS3 style is `border-radius`. Today it's safe to use the standard, nonprefixed style. However, older browsers, such as Safari on iOS 4, needed a `-webkit-border-radius` definition to make it work. Similarly, older Firefox versions required `-moz-border-radius`. So, back in 2010, we needed to add every possible prefixed version in order to ensure compatibility, but today we can safely use the standard definition because all browsers with significant market share support it.



When a feature is evolving to a standard, nonprefixed style, future versions of the browser will support both versions (prefixed and nonprefixed) until the prefixed version is removed.

Vendor prefixes have created a new problem on the Web for mobile devices. As Safari on iOS and Android—the top browsers when the mobile HTML5 era started—are WebKit-based, many developers just used the `-webkit-` prefix. Therefore, no Mozilla, Opera, Microsoft, or standard definitions were declared on millions of mobile websites, leading to a discrimination problem.

This problem led to some browsers, such as Opera and Firefox, announcing that from 2012 they would use the `-webkit-` prefix definition if the developer defined no `-o-` or `-moz-` prefixes, respectively. As Bruce Lawson (<https://twitter.com/brucel>), Opera Evangelist, said in .net magazine (<http://bit.ly/10RpF0Q>):

Opera, along with Microsoft and Mozilla, announced at a CSS Working Group meeting that we would support some WebKit prefixes. This is because too many authors of mobile sites only use the WebKit-prefixed version, and not even the standard, unprefixed one, when it is available. This leads to a reduced user experience on Opera, Mobile Firefox and Mobile IE, which don't receive the same shiny effects, such as transitions, gradients and the like, even if the browser supports those effects.



Remember that Opera for Android, from version 14, has changed its engine from Presto to WebKit. Therefore, `-webkit-` prefixes are the native ones now on this platform.

A similar statement was made by Mozilla's web standard lead in an interview published on the A List Apart website (<http://bit.ly/15Oo4Yz>). As yet, Microsoft has announced no plans to support `-webkit-` prefixes.



Even if Firefox may understand `-webkit-` prefixes on some attributes, we should use and test every possible prefix to maximize compatibility and ensure that everything works as expected when using experimental features.

As developers, the solution we can implement is to use the CSS fallback mechanism. We have two kinds of prefix usages in CSS: styles and values.

For prefixed styles, we can just define all the possible prefixes, and every browser will just take the right one and ignore the others as unknown values. For example:

```
article {
  -webkit-column-count: 2; /* Webkit-based browsers */
  -moz-column-count: 2;    /* Firefox */
  column-count: 2;         /* W3C standard; IE and Opera use it
                             today, and Firefox and WebKit will
                             use it in the future */
}
```

For prefixed values or functions, we just re-declare the same style with different values, as we saw before. For instance:

```
#logo {
  background-image: url("logo.png"); // most compatible version first
  background-image: -webkit-image-set(url("logo2x.png") 2x,
                                       url("logo.png") 1x));
  background-image: image-set(url("logo2x.png") 2x,
                              url("logo.png") 1x));
}
```

CSS preprocessors

To manage all the possible prefixes, we can use a CSS preprocessor or dynamic language. Examples of dynamic languages are LESS and SASS (Syntactically Awesome Style-sheets). Both can be defined as CSS-based stylesheet languages with utilities and extensions not available in the standard CSS. Although browsers today do not support LESS or SASS styles directly, we can use a preprocessor tool that will “compile” our LESS or SASS styles to a final, standard CSS.



Remember that when we use dynamic stylesheet languages such as SASS or LESS we always need to precompile them to CSS before deploying our code for mobile platforms.

SASS. SASS (<http://sass-lang.com>) is defined as a CSS metalanguage that will be interpreted or compiled to standard CSS. It has two different syntaxes: the original syntax,

based on the language Haml (using only indentation), and a modern syntax known as SCSS that is more similar to CSS (using braces).

SASS supports *mixins*, a way to define functions that can be applied on our stylesheets to simplify repetitive tasks, such as adding prefixes. For example, we can define the following mixin using SCSS syntax:

```
@mixin column($count) {  
  -webkit-column-count: $count;  
  -moz-column-count: $count;  
  column-count: $count;  
}
```

Then, when we want to use the mixin, we just use:

```
article {  
  @include column(2);  
}
```

The compiler (<http://sass-lang.com/try.html>) is available for Ruby.



Compass (<http://compass-style.org>) is a SASS plug-in that will help you create more compatible CSS3 websites. It includes prefixes for some properties.

LESS. LESS (<http://lesscss.org>) was based on SASS, with a cleaner syntax and some more abilities. It is available for free and is distributed as a client-side processor (not recommended for mobile browsers) and a server-side processor based on Node.js and Rhino. LESS prefixer (<http://lessprefixer.com>) will give you a collection of LESS mixins that you can use to prefixize all your stylesheets.

To illustrate how LESS works, we can define a LESS stylesheet (with the LESS prefixer included) as:

```
article {  
  .column-count(2); // This is a LESS mixin that will be executed  
}
```

When we compile LESS to CSS, if we define a `column-count` mixin function, we will receive complete, standard CSS with prefixes:

```
article {  
  -webkit-column-count: 2;  
  -moz-column-count: 2;  
  column-count: 2;  
}
```



Both LESS and SASS are more complex, interesting, and useful than the examples shown here suggest. If you like the idea and want to find out more, just go to their websites or buy some books on the topic.

-prefix-free

The JavaScript framework *-prefix-free* (<http://leaverou.github.com/prefixfree>) is an open source solution to the prefix problem. It identifies the current browser and automatically adds all the necessary prefixes for you, on the fly.



There are plug-ins available for *-prefix-free* that allow you to detect changes in stylesheets on the fly and to use the jQuery CSS function without prefixes.

The main advantage of using this technique instead of a server-side or static solution is that you are not sending a big CSS file over the network to be parsed by the browsers. And the big disadvantage is that you are relying on JavaScript for your styles to be applied, which causes a small performance penalty while the page is being loaded.

To use this framework, just download the JavaScript file, include it in your HTML, and write your CSS without prefixes. That's it!

Static tools for prefixes

There are some tools that you can use before deploying your website. That is, you can code with no prefixes or using the prefix you want, and then pass your stylesheets through these tools, which will give you the final code you should use with all the prefixed versions.

The best of these tools are:

- Prefixr (<http://prefixr.com>)
- prefixMyCSS (<http://prefixmycss.com>)
- CSSPrefixer (<http://cssprefixer.appspot.com>)

JavaScript Fallbacks

With JavaScript APIs everything looks simpler, as we can use simple conditionals to verify whether an API, attribute, or feature is available.

JavaScript APIs are usually implemented as:

- Constructor functions, such as `new WebSocket()`

- Functions on standard elements, such as `window.openDatabase()`
- Properties or styles on standard elements, such as `HTMLElement.style.transform`
- Event names, such as `touchstart`

Are you there?

For constructor functions, object functions, and properties, we can just check if they are there—are not undefined—before trying to use them. We can also use the “false” value of an undefined declaration in JavaScript, so the following examples are equally valid:

```
if (navigator.geolocation!=undefined) {
    // Geolocation is available
} else {
    // Fallback
}

if (window.openDatabase) {
    // SQL API is available
} else {
    // Fallback
}

// We can check on property existence
var isGeolocationAvailable = 'geolocation' in navigator;

// Or we can use the double exclamation mark trick to receive
// a Boolean value
var isSQLAvailable = !!window.openDatabase;
```

Using the double exclamation mark (the not not operator), we can force any value to convert to a Boolean value: if the property exists it will return true and if not it will return false.

When working with event names, we can bind to them without worrying about incompatibilities. If the event is not available it will never be triggered, but no error will be raised.



If you are testing against a global function or variable, such as `openDatabase`, always use explicit object syntax (`window.openDatabase`) or your conditional will create an error.

Vendor prefixes

Vendor prefixes are also available in JavaScript. While it’s not as common as in CSS3, some properties, event names, and CSS style declarations are prefixed. While CSS uses dash, prefix name, dash, such as `-moz-`, JavaScript can’t use dashes inside property

names. Therefore, in JavaScript prefixes are just `webkit`, `Moz`, `ms`, or `O`, without dashes and following camel case. The Mozilla and Opera with Presto engine prefixes use uppercase for the first letter, WebKit browsers usually accept either, and Microsoft uses lowercase.

Therefore, for an imaginary prefixed property `hologram`, the prefixed versions should be `webkitHologram`, `MozHologram`, `OHologram`, and `msHologram`.



CSS prefixed styles are bound to prefixed JavaScript attributes on the style property with the same rules; therefore, `-webkit-mask` is bound to `element.style.webkitMask`.

Using the idea of undefined properties having a false Boolean value, we can simplify the problem if every vendor-specific version of the API or property uses the same syntax with the following pattern:

```
var hologram = window.hologram || window.webkitHologram ||  
               window.MozHologram || window.OHologram ||  
               window.msHologram || false;  
if (!hologram) {  
    // API is not supported  
}
```

The engine will take the first nonundefined attribute, or it will just select the last `false` as the value. If you are providing a polyfill, you should replace the `false` with it.



When unifying prefixed properties or functions under one name, be sure that all implementations are using the same syntax. You don't need to use any possible prefix, as some APIs are available only on some platforms.

Modernizr

Modernizr (<http://modernizr.com>) is a free and open source JavaScript library that will detect whether certain HTML5 and CSS3 features are available in the user's browser and expose the results as Boolean JavaScript flags and CSS classes.

When you download the framework you can select one by one all the features that you want to detect, so you get a customized build (Figure 9-3).



Figure 9-3. Modernizr allows us to select which features of HTML5 and CSS3 we want to detect to create our own customized build.



On the desktop side, Modernizr adds support for semantic HTML5 elements, such as section in IE6.

When you include the framework, after the page loads you will receive a Modernizr global object with Boolean flags for every feature checked. For example, if you want to know if CSS columns are available you can use the Boolean attribute `Modernizr.csscolumns`.

Besides the JavaScript flags, the framework adds classes to the body element for each selected feature. If the feature is available, the JavaScript property name is added as a class name. If the feature is not available, a no-`<name>` class is added. For example, for CSS columns, the class name added would be either `csscolumns` or `no-csscolumns`.



Besides all the Modernizr CSS classes, the framework adds the `no-js` class to the body when JavaScript is not enabled in the current context.

As all the website content is inside the body element, we can make use of cascade selectors in the same stylesheet to define different styles, as in:

```
// This style will be applied to columns-compatible browsers
.csscolumns article {
  column-count: 2;
  padding: 10px;
}

// This style will be applied to non-columns-compatible browsers
.no-csscolumns article {
  padding: 5px 50px;
}
```



Modernizr does not remove the need to prefix features. However, there is an optional feature that you can add to your build that will give you a tool that will prefix an attribute for you, as in:

```
var propertyName = Modernizr.prefixed('column-count');
```

It can also look for a prefixed object in a container, as in:

```
var raf = Modernizr.prefixed('requestAnimationFrame', window);
```

Conditional loading

Modernizr adds (as an optional feature) a conditional resource loader that will help you load JavaScript files based on current compatibility. The basic syntax is:

```
Modernizr.load({
  test: Modernizr.feature,
  yep : 'feature-yes.js',
  nope: 'feature-no.js'
});
```

With this tool you can load different JavaScript files based on feature compatibility (replace *feature* with any of the properties that Modernizr supports). The feature can be any Boolean attribute, so we can mix more than one feature, as in `test: Modernizr.geolocation && Modernizr.touch`. The `yep` and `nope` attributes accept one string or an array of strings to load multiple files.

Besides `yep` and `nope`, we can also define scripts with both that will always be loaded, and a complete callback that will be executed after all scripts have been loaded and parsed.

All the properties

The properties that we can query with Modernizr up to version 2.6 are listed in the following subsections. Remember that you can customize your build to contain only the tests you want to use, and more properties can be added as plug-ins. Later in this book, I'll mention some of these properties that can be detected using Modernizr.



Some properties have a subset of properties that we can query on, such as different form input types or video codecs. These properties are not added to the body element as CSS classes; they only appear in the list of JavaScript attributes, such as `Modernizr.inputtypes.date`.

We can use the following feature names to check for compatibility, as properties in JavaScript and as class names that are applied as-is if compatible or with a `no-` prefix if not.



Modernizr uses JavaScript tests to define compatibility. Some false positives are known to occur, and there may be others that have not been identified as yet.

HTML5 main features

- `applicationcache`
- `canvas`
- `canvastext`
- `hashchange`
- `history`
- `input` (subtypes available)
- `inputtypes` (subtypes available)
- `audio` (subtypes available)
- `video` (subtypes available)
- `indexeddb`
- `localstorage`
- `postmessage`
- `sessionstorage`
- `websockets`
- `websqldatabase`
- `webworkers`

CSS main mobile features

- `backgroundsize`
- `borderimage`

- `borderradius`
- `boxshadow`
- `mediaqueries`
- `multiplebgs`
- `opacity`
- `overflowscrolling`
- `textshadow`
- `cssanimations`
- `csscolumns`
- `generatedcontent`
- `cssgradients`
- `cssreflections`
- `csstransforms`
- `csstransforms3d`
- `csstransitions`

Other features

- `geolocation`
- `inlinesvg`
- `smil`
- `svg`
- `svgclippaths`
- `touch`
- `webgl`
- `devicemotion`
- `deviceorientation`
- `fileinput`
- `getusermedia`
- `notification`



If you want to see a complete list of the features that Modernizr can test and the compatibility results on your own device, just visit the Modernizr Test Suite (<http://modernizr.github.com/Modernizr/test>) or haz.io.

To help you understand how it works, if we go back to the last chapter and try to get an idea of form feature compatibility, Modernizr will give us flags on:

- autofocus attribute support: `input.autofocus`
- placeholder attribute support: `input.placeholder`
- required attribute support: `input.required`
- The date/time picker: `inputtypes.datetime`
- The date picker: `inputtypes.date`
- The range slider: `inputtypes.range`
- The HTML5 form validation API: `formvalidationapi`



Some features are available as extensions. To use an extension you just need to download a JavaScript file from Modernizr's GitHub (<http://bit.ly/YbkxQt>).

Polyfills

A polyfill is a JavaScript framework that will replace a feature that is not currently available on a browser with an alternative that will emulate the same syntax as the standard. Therefore, with a polyfill loaded we can use the same standard code even on a browser that doesn't support it.



The Modernizr team maintains a good, updated list of polyfills (<http://mobilexweb.com/go/polyfills>) available for every feature.

Polyfills are not always a good idea on mobile browsers, as we discussed earlier, because of the performance overhead they add to (usually older and less-capable) mobile platforms.

Platform Detection

JavaScript has a native `navigator` object that represents the client browser on which the code is running. We'll take a look at server-side detection shortly, but for now, we can use this technique to detect what device our code is running on and make a decision about what to show based upon that.

The `navigator` object has many properties, but the most useful are `appName` (the browser's name), `appVersion` (the browser's version), `userAgent` (a long string identifying the browser), `plugin` (an array of supported plug-ins for the object tag), `platform` (the operating system), and `userLanguage`.

Generally, we will use the string function `indexOf` to verify whether some of these attributes have the values we are looking for. We can also use `match` and a regular expression. For example:

```
// Detects if it is an Android device
var isAndroid = navigator.platform.indexOf("android")>=0;

// Detects if it is an iOS device
var isIOS = navigator.userAgent.match(/iPhone|iPad|iPod/);
```

The big problem here is that there is no standard. Most browsers will just give you a Netscape value for the name, for compatibility with browsers from the '90s. Table 9-1 gives you an overview of what you get in the `navigator` object for each browser. In this table, assume that `<UA>` will be replaced with each device's user agent ID.

Table 9-1. JavaScript navigator object properties compatibility table

Browser/Platform	appName	appVersion	Strings to identify in the User Agent	Platform
Safari on iPhone and iPod touch	Netscape	5.0(<UA>)	iPhone, iPhone OS	iPhone or iPod
Safari on iPad	Netscape	5.0(<UA>)	iPad	iPad
Android Browser	Netscape	5.0(<UA>)	Android and Mobile	null or Linux armv7l
Chrome for Android	Netscape	5.0(<UA>)	Chrome and Mobile	Linux armv7l
Firefox	Netscape	5.0 (Android/)	Firefox and Mobile	Linux armv7l
Nokia Browser for Symbian	Netscape	5.0 (<UA>)	Symbian and Nokia Browser	S60 or Symbian
Nokia Browser for S40 before 6 th Edition	Nokia	Empty string	Series40	Undefined
Nokia Browser for S40 from 6 th Edition	Netscape	2.0	Series40	Nokia_Series40

Browser/Platform	appName	appVersion	Strings to identify in the User Agent	Platform
Nokia Xpress browser	Netscape	5.0 (X11)	OviBrowser or NokiaBrowser	Linux x86_64
webOS	Netscape	5.0 (<UA>)	wOSBrowser	webOS
BlackBerry browser for smartphones <= 7.1	Netscape	<Platform version> or 5.0 (<UA>)	RIM	BlackBerry
BlackBerry browser for PlayBook	Netscape	5.0 (<UA>)	Tablet OS	BlackBerry
BlackBerry 10 browser	Netscape	5.0 (<UA>)	BB10	BlackBerry
Internet Explorer on Windows Mobile	Microsoft IE Mobile	Empty string	Windows CE	WinCE
Internet Explorer on Windows Phone	Microsoft Internet Explorer	5.0 (<UA>)	Windows Phone OS	Win32
Internet Explorer on Windows 8	Microsoft Internet Explorer	5.0 (<UA>)	Windows NT 6.2	Win32
Opera Mobile (Presto engine)	Opera	<engine version>	Opera Mobi	Symbian, Android, or Windows
Opera Mini	Opera	<engine version>	Opera Mini	Pike<version info>
Amazon Silk	Netscape	5.0 (<UA>)	Silk	armv7L
UC browser	Netscape	UC/<version> (<UA>)	UCBrowser	Empty string

Detect Mobile Browsers

If you want to detect whether your code is running on a mobile browser or not, the simple way is to look for some keywords in the User Agent, such as `mobile` or well-known mobile OS's names. There is a free open source framework that can help you with this—available as JavaScript code and many server-side scripts—from Modernizr (<http://bit.ly/Wf44wB>).



A big question is whether a tablet should be considered a mobile device. Tablet browsers don't usually use the word “mobile” inside the User Agent string, which is a clue that you can provide a more desktop-like version.

Server-Side Detection

Before we get into detection of mobile devices and services on the server, we need to go back a bit and consider an old friend: the HyperText Transfer Protocol, also known as HTTP. Knowing a bit about its internals will help us determine what we can do in terms of mobile web development.



There are no special server requirements for mobile websites; you can just use the same Apache, Internet Information Server (IIS), or other server you are currently using for desktop websites.

HTTP

HTTP is a protocol originally defined in 1991 for document transportation over TCP/IP networks. It has two main versions: 1.0 and 1.1 (the last and current version of the protocol, defined in 1996). This same protocol is the one that we need to use from the server side in mobile web development.

Actually, that last sentence isn't strictly true if we consider WAP 1.1, where the device communicates with the WAP gateway and the WAP gateway is the one connecting to our server via HTTP. A similar approach is used in proxied browsers, like Opera Mini and the Nokia Xpress browser (see Figure 9-4). However, from the server's point of view the requests coming in will always be HTTP requests.

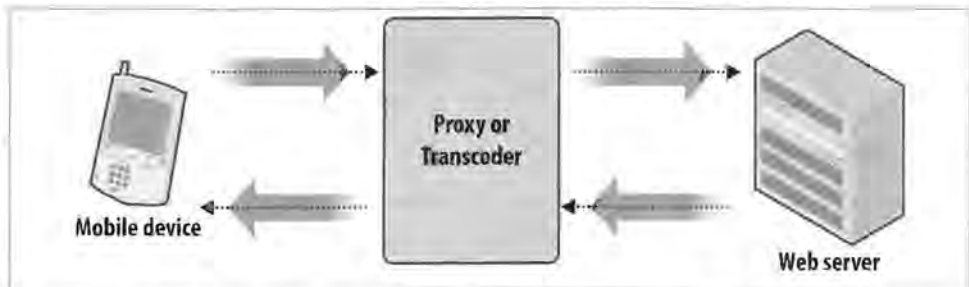


Figure 9-4. When the user is accessing our website via a cloud-based browser or a transcoder, we will not receive the request directly from the user's mobile device.

The request

An HTTP request involves a client (the browser) sending a request to a server using its IP address (previously converted from a domain name). That request has a header and an optional body. The body is generally sent when we are doing a POST request. The most common request type is a GET, requesting a document or a file from the server.

The server responds with a response status code (hopefully not the famous 404), a header, and an optional (but generally sent) body. The body is the requested file.

Why are we taking this two-minute networking class? Because it illustrates many of the techniques we will use in server-side detection.

SPDY and HTTP 2.0

Because HTTP 1.1 was created in 1996 and the Web has evolved enormously over the past few years, Google has developed an open network protocol to replace HTTP, with the particular objectives of reducing page loading latency and improving performance and security. SPDY (pronounced “speedy”) multiplexes and prioritizes all the resources loading through only one connection, maximizing performance without the need of hacks or tricks.

SPDY is based on HTTP, so any HTTP server can work with SPDY if we add a translation layer in between. While SPDY is a trademark of Google, the IETF—the organization behind network standards—is starting work on the proposal of HTTP 2.0, and SPDY will be the starting point.

Google has released an Apache module (<http://code.google.com/p/mod-spdy>) to add support for this protocol. However, at the time of this writing, only big sites such as Google and Twitter are using SPDY on the server side.

As performance is a key issue in mobile browsing, mobile platforms have been among the earliest adopters, with some offering SPDY support from late 2011. The Android browser (from version 3.0), Google Chrome for Android, Amazon Silk, Opera Mobile from 12.1, and Firefox from version 15 support SPDY.

The request header

The request header has many fields that are defined by the browser and sent to the server (if no proxy, gateway, or transcoder is in the middle). Some of the ones that we will find useful are listed in Table 9-2.

Table 9-2. Most common HTTP request header fields

Header field	Description
User-Agent	The name of the browser or platform that originated the request
Accept	A comma-separated values (CSV) list of MIME types accepted by the browser
Accept-Charset	A CSV list of charsets accepted by the browser (e.g., ISO-8859-1, UTF-8)
Accept-Language	A CSV list of preferred languages in the browser
Accept-Encoding	A CSV list of compression methods available for the response (e.g., gzip, deflate)



Every mobile browser supports HTTP authentication, showing a modal window for username and password entry so the user can log in to a website.

The following is the header of a real request from a mobile device to a server:

```
GET / HTTP/1.1
Host: mobilexweb.com
Accept: application/vnd.wap.wmlscriptc, text/vnd.wap.wml,
application/vnd.wap.xhtml+xml, application/xhtml+xml, text/html,
multipart/mixed, */*
Accept-Charset: ISO-8859-1, US-ASCII, UTF-8; Q=0.8, ISO-10646-UCS-2;
Q=0.6
Accept-Language: en
DRM-Version: 2.0
Cookie2: $Version="1"
Accept-Encoding: gzip, deflate
User-Agent: Nokia5300/2.0 (03.50) Profile/MIDP-2.0
Configuration/CLDC-1.1
x-wap-profile: "http://nds1.nds.nokia.com/uaprof/N5300r100.xml"
```



Many mobile browsers send */* as the list of accepted MIME types to avoid the server prefiltering the content that it can deliver using the Accept header. Other mobile browsers have known bugs in the MIME type lists that they provide.

The user agent

The user-agent string identifies the browser. It has had a complex history, with the result that today there are browsers that identify themselves as six different browsers at the same time in the same string. This somewhat complicates browser detection. There is an excellent history of the user-agent string, presented in a very funny way, at We-bAIM (<http://bit.ly/Yd6Sqo>).

In brief, in the beginning of the web era, developers often looked for a particular string in the User-Agent header to determine what content to deliver, many browsers—starting with Microsoft Internet Explorer—started using a hack that today has resulted in user agent hell. The hack was for Internet Explorer to identify itself as Mozilla (the way that the Netscape browser, IE's main competitor, was identified). After this initial identification, it clarified that it was not actually Mozilla, but rather a compatible browser (IE). Microsoft also added other information to the user-agent string, like the operating system and details on the plug-ins and languages supported. The end result was a very complex user agent syntax with no standards.

In the mobile world, the situation is even worse. Some browsers use the IE hack and identify themselves as Mozilla (with some clarification), others identify themselves with

the correct browser name, and still others send the device brand and model number in the User-Agent header. The same device may even provide a different user-agent string, depending on the OS version or the firmware used. This makes device detection using this string a bit complex.



If you don't want your site to be transcoded, you can use Cache-Control: no-transform both in the HTML document and in the HTTP headers. Some transcoders use this information to decide whether or not to transcode a document, but there is no guarantee that this will work.

The following is a list of some mobile user-agent strings provided by a Nokia N95, a Nokia 3510, a Samsung Galaxy SIII, a BlackBerry, an iPhone 5, a Windows Phone 8 device, a Nexus 7, and a Japanese phone from the carrier Au:

- Mozilla/5.0 (SymbianOS/9.2; U; Series60/3.1 NokiaN95/20.0.015 Profile/MIDP-2.0 Configuration/CLDC-1.1) AppleWebKit/413 (KHTML, like Gecko) Safari/413
- Nokia3510i/1.0 (05.30) Profile/MIDP-1.0 Configuration/CLDC-1.0
- Mozilla/5.0 (Linux; U; Android 4.0.4; en-gb; GT-I9300 Build/IMM76D) AppleWebKit/534.30 (KHTML, like Gecko) Version/4.0 Mobile Safari/534.30
- BlackBerry8100/4.2.0 Profile/MIDP-2.0 Configuration/CLDC-1.1 Vendor ID/125
- Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/6.0 Mobile/10A405 Safari/8536.25
- Mozilla/5.0 (compatible; MSIE 10.0; Windows Phone 8.0; Trident/6.0; IEMobile/10.0; ARM; Touch; NOKIA; Lumia 920)
- Mozilla/5.0 (Linux; Android 4.1.1; Nexus 7 Build/JR003D) AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.166 Safari/535.19
- UP.Browser/3.04-TS14 UP.Link/3.4.4

As you can see, there is no standard for this string. So, the initial approach is to search for some term—for example, “iPhone” or “Android”—to try to determine which device or platform a request has originated from. Don't worry if this sounds nightmarish, though; in a few pages we will talk about a better solution.

iPad or iPhone?

When the iPad (the Apple tablet) came to the market, many thought that servers would automatically redirect iPad users to the iPhone versions of websites, if they existed. That is because the iPad uses iOS, just like the iPhone itself and the iPod touch.

However, the iPad User-Agent header looks something like this:

```
Mozilla/5.0 (iPad; CPU OS 6_0 like Mac OS X) AppleWebKit/536.26 (KHTML,  
like Gecko) Version/6.0 Mobile/10A5355d Safari/8536.25
```

As you can see, it doesn't contain the word "iPhone," and that is one of the ways that developers detect usage of iPhone-based devices. It does include the word "Mobile," though, so if you are looking for that word you will be able to tell that the user is on a mobile device.

That said, because the iPad has a larger screen and can show more content than a non-tablet mobile device, mobile websites are not ideal. Some big sites have created special iPad versions using Safari on iOS extensions; these are neither the mobile nor the desktop web versions, but somewhere in between.

The iPod touch does include "iPhone" in the user-agent string because it is so similar (in terms of the operating system and screen size) to the iPhone itself.

What we can identify

Identification is useful for context definition. We need to remember that the user is a mobile user, and for these users the context is very important. We want to get all the information we can about that context, so we can provide the most useful experience possible.

The mobile browser doesn't send information about:

- The International Mobile Equipment Identity (IMEI) or serial number to identify the device uniquely
- The type of network used (WiFi, 3G, GPRS, EDGE, CDMA)
- The carrier (operator) providing service to the device
- The country of the user
- If the user is roaming
- The phone number of the user
- The device's brand and model number (at least, not directly)

Some of this information (carrier, brand, and model number) can be inferred, but the other identification data will not be available. That is why we cannot identify users automatically without a login, as in desktop web applications.



If you are working closely with the carrier for your mobile website, you may be able to have yourself added to the WAP gateway's URL whitelist. You will then be able to receive a customer ID or phone number in a new, nonstandard header.

Here's what we can glean from the device headers:

- The carrier and country, from the IP address of the request (if it is using a wireless network, such as 3G or 4G)
- The country (and maybe city or even exact location), from the IP address of the request (if it is using a WiFi network)
- The brand and model number, inferred from the User-Agent header
- The language in which the operating system is defined
- What markups and document types are accepted, if the header is not defined as `*/*`

The iOS Detection Problem

The user-agent string on iOS devices includes the operating system version (such as 5.1 or 6.0) and the device type (such as iPod, iPhone, or iPad), but it doesn't include any reference to the current device hardware or generation. Therefore, there is no way of using HTTP headers to determine if the user is browsing with an iPhone 3GS, an iPhone 4, or an iPhone 5. Likewise, you can't tell if the user has a first- or third-generation iPad.

Apple has done this intentionally to avoid developers trying to create different experiences per generation of device. Up to iPhone 5 it was not a big problem, as all the devices in a given category (all the iPhones, all the iPads) were quite similar in terms of how websites intended for viewing on those devices should be. On the iPhone 5 and fifth-generation iPod touch, the larger screen may require some attention, but if you design your website with good practices, you should not have any problem. Anyway, on the server side there is no way to tell if the user is on a larger screen device.

The problem continues with the iPad mini, as it exposes the same user agent as other iPads with the same iOS versions—there is no way to detect if the user is on a 7.9" or 9.7" iPad.

The inability to differentiate between hardware generations is also a problem for statistics, as we don't know how many visitors to our websites are on newer devices.

The only trick we can apply is to use a JavaScript solution that will look into the pixel ratio and screen dimension values to make assumptions about the type of device, and store that information in a cookie for later visits (or use Ajax to send that information to the server). This client-side trick can't be applied to the iPad mini, though, as every CSS and JavaScript property on this device exposes the same values as on the iPad 2.

The User Agent Profile

The *UAProf* (User Agent Profile) is a voluntary standard defined by the Open Mobile Alliance (formerly WAP Forum). It takes the form of an XML file defining the abilities of the device, including its screen size, download features, and markup support.

The XML is defined by the manufacturer or the carrier, and a link to the XML is defined in a header (typically `x-wap-profile`). If you look back at the sample headers we saw earlier in this chapter, you'll see it:

```
x-wap-profile: "http://nds1.nds.nokia.com/uaprof/N5300r100.xml"
```

This URL will have been defined when the browser was created, which may have been several years before. If the URL is still working, we should get XML like the following extract:

```
<prf:ImageCapable>Yes</prf:ImageCapable>
<prf:Keyboard>Qwerty</prf:Keyboard>
<prf:Model>BlackBerry 8100</prf:Model>
<prf:NumberOfSoftKeys>0</prf:NumberOfSoftKeys>
<prf:PointingResolution>Character</prf:PointingResolution>
<prf:PixelAspectRatio>1x1</prf:PixelAspectRatio>
<prf:ScreenSize>240x260</prf:ScreenSize>
<prf:ScreenSizeChar>26x18</prf:ScreenSizeChar>
```

UAProf files have many problems. First, we need to download the XML for each request, process it, and either extract the properties we want or make a local copy on our server. Second, the community has found lots of bugs and problems in this official information. Some devices (the iPhone, for example) don't define a UAProf file, and they don't all define the same properties. Also, UAProf does not have the right granularity of information; for example, you can read that Flash is supported, but no information is there about which version.

So, another problem added to our list.



Some mobile browsers, mostly on smartphones and tablets, support a feature called something like “Request desktop site.” When the user enables this feature, the browser changes its user agent to a desktop browser user agent. The result is that on the server side there is no way to tell that the user is on a mobile device, so you will most likely deliver a desktop experience.

User agent tricks

As we already know, there are different versions of Firefox available for Android, MeeGo, and Firefox OS. However, while the user-agent strings for Firefox reveal which platform the user is on, they do not identify the brand and model of device the user is using to browse the Web.

Opera Mobile had a similar problem, but starting with version 12.1 it includes a `Device-Stock-UA` header whose value is that of the user agent of the default browser on the current device, so we can detect which device it is regardless of which Opera version the user has. The `User-Agent` header doesn't include any reference to the device hardware.

On Windows 8 we can identify if the user is on a touch device by looking for the `Touch` token inside the user-agent string. We can also tell that it's a Windows 8 tablet if the `ARM` string is there; if not, it may be an Intel-based tablet, a notebook, or a desktop, and we can use `Touch` to figure out if it's a touch device. Microsoft does not indicate the brand and model of the device anywhere in the user-agent string.

Web views, such as native web apps, usually browse the Web with the default web browser user agent. However, on some platforms, developers can customize those user agents. For example, Chrome for iOS adds a `CriOS` token to the default Safari on iOS user agent.



For browsers that are available on different device types, such as tablets and smartphones, usually the way to differentiate them is to look for the `Mobile` token for smartphones and the `Tablet` token for tablets. This rule applies to Firefox, the BlackBerry browser, and Opera. For the Android browser and Chrome, you know the user is on a tablet if the `Android` token is there and the `Mobile` token is not. For iPads, you just look for the `iPad` substring.

Detecting the Context

We've already seen how the network protocol works, and what information is provided and not provided in a mobile browser request. Now let's get some data and information from the context.

How to read a header

The specifics depend on the language, but all server-side platforms offer a way to read the request's header. Some languages use the parameter name as the header name (for example, `Accept-Charset`), and others use a longer version with the syntax `HTTP_X`, where *X* is the header name in all uppercase and with the dashes (-) replaced by underscores (for example, `HTTP_ACCEPT_CHARSET`).



Some Nokia devices expose a custom HTTP header that defines the connection type. The header is `x-nokia-musicshop-bearer` and the possible values are `WLAN` or `GPRS/3G`. We can try to read this header and, if it exists, get more information about the context.

In Java Servlets or JSP, we read a header using:

```
request.getHeader("header_key")
```

And in PHP:

```
$_SERVER["header_key_large"]
```

In ASP.NET with C# or Visual Basic, we have a `Headers` collection and public members for most of the common headers:

```
// This is the C# version  
Request.Headers["header_key"]
```

```
' This is the VB version  
Request.Headers("header_key")
```



Older BlackBerry devices expose a custom HTTP *via* header that can be used to see which browser is being used. For example, if the value contains an `MDS` string, the user may be connected via the BlackBerry Browser through the corporate server; if it contains `BISB` the user is using the Internet Browser connecting through the carrier; and if the value is not defined, the user may be using the WiFi Hotspot Browser.

How to read the IP address

The IP address from which the request originated can be read with the following code:

```
// In Java  
String address = req.getRemoteAddr();
```

```
// In PHP  
$address = $_SERVER["REMOTE_ADDR"];
```

```
// In C#  
String address = Request.UserHostAddress;
```

What can we do with the IP address? The next chapter will talk about geolocation. However, if we want to define the user's carrier and country right now, we need to get an updated list of the IP ranges assigned to each carrier. The carriers distribute this information to their partners, and it can also be found in forums and communities or through commercial services.



Massive's Operator Identification Platform is a community-based database service that allows us to determine visitors' countries and network operators, if detected, using a simple HTTP service request. You can request an account from massive (<http://www.werwar.com>).

Cloud-Based Browsers

As mentioned earlier, there are some cloud-based browsers on the market (Opera Mini is the most widely installed), and we need to take care of differences in the headers in such browsers. Even on a well-known device, such as a BlackBerry or an iPhone, if Opera Mini is in use the requests we receive on our servers will come from the Opera Mini proxy and not from the device itself. So, the client IP address will be Opera's server address, and the user-agent string will be the proxy's one.



Remember that cloud-based browsers are not just something for feature phones or older devices. Modern platforms can also browse the Web using these browsers, such as Opera Mini on iPhone/iPad or Android devices, Chrome on Android, the Nokia Xpress browser on Windows Phone devices, or Silk on an Amazon Kindle Fire.

Opera Mini

On any device, the Opera Mini user-agent string looks like this:

```
User-Agent: Opera/9.80 (J2ME/MIDP; Opera Mini/6.1.25378/25.692; U; en)  
Presto/2.5.25 Version/10.54
```

The substring between the open parenthesis and the first semicolon is the device's platform.

Possible options are:

- Android
- BlackBerry
- BREW

- J2ME/MIDP
- iPhone
- iPad
- MTK
- Series 60
- Windows Mobile

Fortunately, Opera Mini offers the original IP address and the original user-agent string, along with other information, in new headers (listed in Table 9-3) that we can read using the techniques we have already seen.



Other cloud-based browsers are not as developer-friendly as Opera Mini and remove all the original headers, so we are blind in terms of detecting the device and its origin.

Table 9-3. Opera Mini additional optional HTTP headers

Header	Description
X-OperaMini-Phone-UA	Provides the user-agent string identifying the device that downloaded the Opera Mini client (or the current device's user-agent string if not available).
X-OperaMini-Phone	Provides the device's brand and model, separated by a hash (<code><brand>#<model></code>).
X-Forwarded-For	Provides a CSV list of all the proxy servers in the chain that have forwarded the request from the device to Opera Mini's proxy. Opera recommends using the last IP address listed for geolocation purposes.
X-OperaMini-Features	Provides a CSV list of phone features, from the following list: • basic (Java MIDP 1.0 device, low resources, almost nonexistent today) • advanced (Java MIDP 2.0 device, high resources) • camera (camera detected, so we can provide a file upload input for pictures) • file_system (filesystem support detected, so the user can download and upload files) • download (the device supports download files) • touch (the device has a touch screen) • viewport (the viewport meta tag is compatible with the device) • secure (connection between phone and proxy is encrypted)

Xpress browser

The Xpress browser is the cloud-based browser available for Nokia Series 40 and Windows Phone devices. Sometimes the browser will use the `NokiaBrowser` or `OviBrowser` identification, as it was also known as the `Nokia Browser` or `Ovi Browser` on older devices. `Novarra` is also used a lot inside headers, because this browser was developed as a result of Nokia's acquisition of Novarra, Inc. (a company that used to create transcoders and proxies for mobile web).

The specific HTTP headers the Xpress browser supports are listed in Table 9-4.

Table 9-4. Nokia Xpress browser additional HTTP headers

Header	Description
X-Mobile-Gateway	Specifies the proxy server version: for example, Novarra-Vision/8.1.
X-Forwarded-For	Provides a CSV list of all the proxy servers in the chain that have forwarded the request from the device to the cloud's proxy. The last IP address (or the only value, if there's only one) is the user's IP address.
X-Nokia-Device-Type	Undocumented feature; in tests it's always zero (0).
X-WAP-Profile	Provides the URL to the current device's User Agent Profile (forwarded from the original request from the device; not available on all devices).

Amazon Silk

The Kindle Fire browser does not add any additional headers when the acceleration mode is enabled and the Amazon Cloud is between your server and the user. However, the Silk user agent will have the string `Silk-Accelerated=true` at the end if the acceleration mode is enabled, indicating that it's using the cloud-based acceleration proxy. If not, it's acting like a direct browser.



If you are using a server-side detection to provide different layouts, you can try them with a User Agent Switcher plug-in for desktop browsers or use the online tool Hydra Multi-Device Proxy (<http://hydra.mobiforge.com>).

Mobile Detection

If you only want to know whether the user is browsing from a desktop or a mobile device (perhaps for doing a redirection), the quickest way to find out is to check for some different well-known strings (iPhone, iPod, Nokia) inside the User-Agent header. Based on their presence or absence, you can make an educated guess about whether or not the user is on a mobile device.



There is an excellent collection of mobile-specific User-Agent headers at mobiForge (<http://bit.ly/YtOvwX>).

Andy Moore has developed a very simple but powerful PHP script for detecting mobile user agents and browsers. The latest version (free for nonprofit purposes) can be downloaded from Detect Mobile Browsers (<http://detectmobilebrowsers.mobi>). You can also

define what to do in specific situations, such as whether or not to consider iPads mobile devices.

A simple but effective script that will look into dozens of substrings inside the user agent is available from Detect Mobile Browsers (<http://detectmobilebrowsers.com>). You will find scripts for PHP, ASP.NET, ColdFusion, JSP, Node.js, Perl, Python, and Rails. You can also download configuration files for Apache or IIS.



Some older devices used to support multipart document delivery. A multipart document includes XHTML and resources (images, CSS) in the same HTTP response, enhancing the download performance. To determine whether a device supports multipart documents, check for the `multipart/mixed` or `application/vnd.wap.multipart` MIME type in the Accept header. SPDY and HTTP 2.0 came to use the same idea on modern websites.

Transcoders

With WAP 1.1, operator gateways were required to precompile WML and make it lighter and more easily parsable by devices with limited memory/CPU resources. Gateways were also relied upon to manage cookies on behalf of the devices (which did not have enough memory) and for integration with the operator's backend (for example, the gateway was able to inject the users' phone numbers into HTTP headers, so that authorized content providers could recognize them and bill them for services).

With WAP 2, precompilation was no longer needed, but other aspects of gateways were, and still are today, to some extent. In a lot of different contexts, the presence of a WAP gateway is beneficial to developers and content providers.

Around 2002, some companies started selling tools to “mobilize” web content. That is, users could type a URL into a field, and the transcoder/content reformatter would chop it up into pages that could be viewed by mobile devices.

Around 2006, some kind of genetic mutation happened: transcoder vendors realized that transcoders could be deployed in proxy mode and the whole Web could be transcoded behind the backs of the users and content providers, regardless of the presence of a mobile-optimized experience for any given site. This dangerous move posed a serious threat to the mobile ecosystem.

Hopefully, in the HTML5 era transcoders will not be used as they were before. If you want to know more about transcoders, check out www.mobilexweb.com/go/transcoders.

The W3C has released a couple of guidelines (<http://www.w3.org/TR/ct-guidelines>) for transcoders to follow. The community of mobile developers on the WMLProgramming

list have also created their own guidelines, known as the Manifesto for Responsible Reformatting (<http://wurfl.sourceforge.net/manifesto>).



The official HTTP way to define that we don't want our content to be transformed is to add the following header in our HTTP responses: `Cache-Control: no-transform`.

Device Libraries

As we've discussed, just looking at the HTTP headers and the UAProf will not give us enough useful information about the mobile devices that are accessing our websites. This is where device libraries come to our assistance. Device libraries are offline databases (or cloud-based services) that take a user-agent string (or all of the request headers) and return to us dozens of properties about the detected device, from screen size, to HTML5 compatibility, to Ajax support and video codec compatibility.



Device libraries are also known as Device Description Repositories (DDRs).

The most well-known libraries available today are WURFL, DeviceAtlas, and Detect-Right, but we'll also look at a few other projects, including 51Degrees.mobi and OpenDDR.

WURFL

The Wireless Universal Resource File (known as WURFL) is an open source device capabilities repository maintained by ScientiaMobile (<http://www.scientiamobile.com>) under a dual AGPL v.3 and Commercial license. It also includes APIs for the major server-side platforms and a cloud-based service.

The WURFL Framework consists essentially of two components: an API and the repository in XML format. The API is dually licensed (AGPL/Commercial), while the repository is distributed with a license that only allows usage with the APIs released by ScientiaMobile. The AGPL license allows free usage of the software only as long as one is willing to open source the entire application, irrespective of the fact that the application may be hosted on a web server. ScientiaMobile has also launched a WURFL Cloud Service, which includes a free-of-charge offering for hobbyists and small projects.



Before 2011, WURFL was a free and open source project. According to Luca Passani, CTO of ScientiaMobile and creator of the project, WURFL had reached the point where further improvements were not possible without the injection of development resources and this was the driver of the new commercial open source approach.

If you have been using WURFL for a while, you should check the new license conditions. There is a version of the cloud-based service you can still use free of charge.

The WURFL APIs can be downloaded from the WURFL website (<http://wurfl.sourceforge.net>). Repository updates are made available to commercial licensees on a weekly basis and community (AGPL) users are offered quarterly snapshots.



The AGPL v.3 license allows free usage if you are ready to open source your code. If not, you should acquire a commercial license from ScientiaMobile.

The sources of information include official technical information published by manufacturers, UAProf files, and data collected by the community and the company after testing on real devices. Today, the WURFL database contains thousands of devices (15,000 profiles, 8,000 of which have a unique brand and model), with information on subversions, operating systems, firmware and hardware variations, and hundreds of attributes that we can query for each one.

WURFL is available as three possible implementations:

- Standalone for web languages (Java, PHP, .NET)
- Standalone for C++ (high performance and integration with network-level software)
- Cloud (Java, PHP, .Net, Perl, Ruby, Python, and Node.js)

With the standalone license (also known as WURFL OnSite), you will be able to install the repository and the API on your own server; you'll need to update the repository frequently on your own but you can use it for free under AGPL v.3. If your site is commercial and you don't intend to comply with the AGPL terms, you can acquire a commercial license.

With the standalone commercial license you will receive:

- Weekly repository updates

- The Java, PHP, Database (previously known as Tera-WURFL), or .NET API



The WURFL InFuze product series includes a C++ API as well as three enterprise-level products based on it: the WURFL Modules for Varnish Cache, Apache Web Server, and NGINX Web Server. These products enable features such as the ability to enrich the HTTP headers or the ENV environment variables with WURFL capabilities, effectively relieving organizations from the need to deploy device detection libraries as part of their applications.

With the Cloud Service, you don't need to install anything on your server with the exception of a light client; you will have a continually updated repository available to you at different monthly rates, and even a free service. The main disadvantage of the Cloud Service is that every time a new user makes a request, your script needs to contact the ScientiaMobile servers and wait for the response before getting the WURFL data and releasing the response to the user.

The Cloud Client comes with enough intelligence to save unnecessary round-trips for devices that have already been detected. Exactly which caching strategies are adopted also depends on the actual WURFL Cloud plan you select. The options are:

Free

Allows you two capabilities of your choice, 5,000 detections per month, and one domain/IP address

Basic (\$10/month)

Allows you five capabilities of your choice, 50,000 detections per month, and one domain/IP address

Standard (\$40/month)

Allows you 10 capabilities of your choice, 2 million detections per month, three IP addresses, and two domains

Premium (\$500/month)

Allows you unlimited capabilities, 10 million detections per month, 25 IP addresses, and three domains

With any plan you will get access to the full list of capabilities that you can detect, and a WURFL Cloud Client available in PHP, Java, Ruby, Python, Node.js, and .NET.

Architecture. WURFL groups the devices into a hierarchy of devices and capabilities. Some devices are equivalent to other devices from the same series, possibly with some new features, so there is a fallback mechanism in WURFL allowing a device to extend

the features of another one. The same applies for different models in the same brand or even different brands with the same operating system.

Also, there is a feature called “actual device root” that manages multiple sub-versions (different firmware) of the same device, so the information is not duplicated in two records; the sub-version will be based on the main record with any added or different abilities noted.

The WURFL database has a root fallback device called “generic device” that is matched when the device, the brand, and the series can’t be determined.



The WURFL repository also includes detection for desktop browsers and smart TV platforms.

Capabilities. Every ability, property, or attribute is called a *capability* in the WURFL world. Capabilities are organized into groups. Each capability for each device has an optional string value (taken from the device itself, or from the fallback mechanism). That value can be converted to a Boolean, a number, a string, or an empty string. You can find an updated list of capabilities at ScientiaMobile (<http://www.scientiamobile.com/wurflCapability>).

The most useful groups at the time of this writing are shown in Table 9-5.

Table 9-5. Most useful WURFL capability groups

Group name	Capabilities related to
product_info	Device information, such as the brand, model, operating system, and browser
xhtml_ui	XHTML rendering, including tel URI scheme support, accesskey support, iframe support, and file upload support
ajax	Ajax and DOM support, including support for getElementById, innerHTML, and CSS manipulation
markup	Markup compatibility
cache	Cache support
display	The screen and display (physical dimensions, resolution, line rows, and so on)
image_format	Image formats, including support for animated GIFs and SVG
wta	WTAI, including voice call support
security	Encryption, including HTTPS support
bearer	Networks, including WiFi and VPN support
storage	Limits (such as max URL length)
object_download	Formats and object downloading support for each typical format
streaming	Audio and video streaming per format and codec

Group name	Capabilities related to
wap_push	WAP Push attribute support
j2me	Java ME configuration and profile versions and API compatibility
mms	MMS support
sms	SMS support
sound_format	Support for audio codecs and formats
flash_lite	Flash support on the browser, for standalone applications, and for wallpaper or screensavers
css	CSS properties
transcoding	Whether the client is detected as a transcoder
rss	RSS support
pdf	PDF viewing support
playback	Formats that can be played by the device
smarttv	Smart TV support

As you can see, the information that the XML provides is really complete. If you want to browse all the capabilities, you can do so at ScientiaMobile (<http://dbapi.scientiamobile.com/explore>).

Table 9-6 shows the most important capabilities we can query, based on the compatibility problems outlined in the preceding chapters. Remember that there are dozens of other properties that you can query; take a look at the library so you'll have an idea of all the possibilities.

Table 9-6. Most useful WURFL capabilities

Capability name	Type	Indicates
brand_name	String	The device's brand name (such as Apple, Nokia, or HTC)
model_name	String	The device's model name (such as iPhone, N97, Nexus One)
marketing_name	String	The device's marketing name, including the brand, model, and possibly another part of the name (Pearl, touch)
ux_full_desktop	Boolean	Whether the device is optimized for a full desktop experience (true) or not
is_wireless_device	Boolean	Whether the device is a mobile device (true) or not
is_tablet	Boolean	Whether the device is a tablet device (true) or not
is_smarttv	Boolean	Whether the device is a Smart TV (true) or not
pointing_method	String	Which pointing method is accepted (joystick, stylus, touchscreen, clickwheel, or the empty string)
device_os	String	The name of the operating system
device_os_version	String	The version of the OS
mobile_browser	String	The name of the browser
mobile_browser_version	String	The version of the browser

Capability name	Type	Indicates
resolution_width	Integer	The screen width in pixels
resolution_height	Integer	The screen height in pixels
max_image_width	Integer	The display's usable width in pixels
max_image_height	Integer	The display's usable height in pixels
xhtml_support_level	Integer	The level of XHTML compatibility, from -1 to 4: -1: No support 0: Basic support (poor or no CSS support, basic form support, basic or no table support) 1 and 2: Advanced basic support (basic CSS and table support) 3: Medium support, including excellent CSS support) 4: Advanced support, including Ajax support
preferred_markup	String	The markup best supported by the device (even if it supports a newer one)
xhtml_format_as_css_property	Boolean	Whether -wap-input-format is available
xhtml_make_phone_call_string	String	The prefix preferred for making phone calls in a URL
xhtml_send_sms_string	String	Whether and how the device supports triggering the SMS client from a link (can be sms:, sms to:, or the empty string, meaning not supported)
xhtml_file_upload	String	Whether the device allows file uploading (returns not_supported, supported, or supported_user_intervention)
xhtml_supports_iframe	String	Whether the device supports iframes (returns none, partial, or full)
ajax_supports_javascript	Boolean	Whether the device supports JavaScript with basic operations (dialogs, form values, timers, and document.location)
ajax_support_event_listener	Boolean	Whether the browser allows event registration through event listeners
html_wi_oma_xhtmlmp_1_0	Boolean	Whether the browser supports XHTML MP 1.0
html_web_4_0	Boolean	Whether the browser supports HTML4
gif_animated	Boolean	Whether animated GIFs are supported
svg_t_1_1	Boolean	Whether SVG 1.1 is supported
svg_t_1_1_plus	Boolean	Whether SVG 1.1+ is supported
flash_lite_version	String	Which version of Flash is supported
fl_browser	Boolean	Whether the browser supports Flash content
is_transcoder	Boolean	Whether a transcoder was detected as a proxy from the real device
transcoder_ua_header	String	Which header we can find the original device's user-agent string in, if a transcoder was detected
viewport_supported	Boolean	Whether the viewport meta tag is supported
image_inlining	Boolean	Whether the data URI scheme is supported to inline images
css_spriting	Boolean	Whether CSS sprites are supported

WURFL Standalone API (OnSite). These APIs allow us to use WURFL in a couple of lines, with many advantages:

- Automatic device detection using the header information
- Two-step user agent analysis (optimized and clever user-agent searching inside the XML)
- Detection of transcoders and proxies, and matching of the correct user agent and device information
- Merging of the static XML provided by WURFL with patches (the web patch or your own), providing a simple and unique way to query capabilities
- In-memory caching of the XML parsing for the best performance on every request



When you have your license activated, you can download snapshots weekly and update them manually on your server, or you can create a direct download link that will be unique for you and will let you download the latest WURFL snapshot (the device and capability database) and automatically apply it on your own server. Just go to My Account and look for the direct download section.

PHP API installation. We are going to use PHP to illustrate the use of the WURFL Standalone API. To begin, you should download the PHP API from ScientiaMobile (<http://www.scientiamobile.com/downloads>) and extract the contents of the ZIP file. The package contains documentation, examples, resources, unit tests, and a *WURFL* folder where the API resides.



The PHP WURFL API allows us to save persistence information (device profiles) and cached request information using memcache instead of using the filesystem. The filesystem was chosen as the default because it maximized users' ability to get the WURFL API to work out of the box, but it's not the preferred option in the long term.

To make it work, follow these steps:

1. Copy the *WURFL* folder into your web server root folder.
2. Copy the *examples/resources* folder into your web server root folder (you can change the name).

3. Optionally, download the latest *wurfl-<version>.zip* file from WURFL (<http://wurfl.sourceforge.net>) and copy it to the new *resources* folder.
4. Verify that PHP scripts have write permissions on the *resources/storage* folder.
5. Edit the *resources/wurfl-config.xml* file and check that the *<main-file>* tag matches the name of the ZIP file containing the main XML repository. It can also be a decompressed XML file.
6. Edit the *resources/wurfl-config.xml* file, go to the persistence and cache nodes, and check that the *<params>* tags match the names of the *cache* and *persistent* folders inside *resources/storage*, as in *<params>dir=storage/cache</params>*. The path needs to be relative to the *config* XML folder.

Once WURFL is installed, we can create our first PHP script that uses the repository. Using version 1.4 of the API, the code will be:

```
<?php
$wurflDir = dirname(__FILE__) . '/WURFL';
$resourcesDir = dirname(__FILE__) . '/resources';
$wurflConfigFile = $resourcesDir . '/wurfl-config.xml';

require_once $wurflDir . '/Application.php';

// Create WURFL Configuration from an XML config file
$wurflConfig = new WURFL_Configuration_XmlConfig($wurflConfigFile);

// Create a WURFL Manager Factory from the WURFL Configuration
$wurflManagerFactory = new WURFL_WURFLManagerFactory($wurflConfig);

// Create a WURFL Manager
$wurflManager = $wurflManagerFactory->create();

// Get device object with all capabilities
$device=$wurflManager->getDeviceForHttpRequest($_SERVER);
?>
```

The first time you run this file on your web server (local or remote), you will need to wait a minute or two while it creates the persistent data to enable quick detection in future requests. If you receive a blank page, great! If you get an error, you need to check all the steps again.



If you're working on a local server and are going to upload your website to another server using FTP or some other protocol, it will be better not to upload the cache folder because it will contain thousands of files. It is better to leave the server to re-create them locally.

Using the PHP API. The PHP API is an object-oriented API. Once you have the WURFL Manager object, you can use it.



If your server is too heavily loaded, you may get a timeout error when processing WURFL the first time. If this happens, ask your server provider how to increase the maximum script time limit or change the *PHP.ini* file.

A typical usage is getting a device object using the manager's methods:

- `getDeviceForHttpRequest($_SERVER)`
- `getDeviceForUserAgent($user_agent)`
- `getDevice($deviceId)`

If you want to access the capabilities of the current device accessing your website, the first option is the best one. If you want to get properties for other devices, you can use the `user_agent` method or the `deviceId` method. Every device in WURFL has an ID that you can store in your databases for statistical or logging purposes. You can then look for its capabilities later, after the mobile request.



You can browse the WURFL device database using the free online tool ScientiaMobile Tera-WURFL Explorer (<http://dbapi.scientiamobile.com/explore>).

You can also get all the possible groups and capabilities using `getListOfGroups()` and `getCapabilitiesNameForGroup(groupId)`, both methods of the manager.

Once you have the device object, you can get all the properties with the `getAllCapabilities()` method or query for one particular feature with `getCapability($capabilityName)`.

If you are using the desktop web patch, you can determine whether the client is a desktop or a mobile device:

```
if ($device->getCapability('is_wireless_device')==false) {
    // It is not a mobile device
    header('Location: http://yourdomain.com');
}
```

To detect if it is an Apple device, use:

```
if ($device->getCapability('brand_name')== 'Apple') {
    // It is an iPhone, iPod or iPad, redirect to a prepared version
    header('Location: http://yourdomain.com/ios');
}
```

With the capability, you can then decide whether or not to provide some feature. For example:

```

if ($device->getCapability('html_file_upload')== 'supported') {
    echo '<input type="file">';
}

```

A great option if you are offering content is to explicitly display to the user his phone model in the marketing information:

```

echo 'Download compatible content for your ' .
    $device->getCapability('brand_name') . ' ' .
    $device->getCapability('model_name') ;

```



If you want to test whether your WURFL code is working on your desktop browser, you can use Firefox and the free plug-in User Agent Switcher that allows Firefox to change its user-agent string to that of any other device of your liking. We will cover this plug-in in Chapter 18.

The WURFL Cloud API. If you go with the Cloud API, you first need to get a key from a free or paid plan from ScientiaMobile (<http://www.scientiamobile.com/cloud>). When you are done, you will get a dashboard page where you can see the status of your account (detections for the month, capabilities you can detect, and so on).

The first step is to create a key and save it. The key is a string that looks like 995405:LU7z6FfYuaaabQVWvPZT8gk5G2qDM5ps.

After that, use the Cloud Capabilities Selector to choose all the capabilities you want to enable (Figure 9-5), and download the appropriate version of the WURFL Cloud Client software for your development environment. We will use PHP for demonstration purposes.



Figure 9-5. Using ScientiaMobile’s WURFL Cloud Service, we can select which capabilities we want to detect, based on our license level—the free service allows us to select up to two capabilities.

When you download the PHP library, you can unzip it in your root folder and then use the following code snippet. Be careful to define where you copied the API as wurfl-cloud in this example. You should copy all the folders, except the examples:

```
<?php
// Include the WURFL Cloud Client
// You'll need to edit this path
require_once 'wurfl-cloud/Client/Client.php';

// Create a configuration object
$config = new WurflCloud_Client_Config();

// Set your WURFL Cloud API key
$config->api_key = 'xxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx';

// Create the WURFL Cloud Client
$client = new WurflCloud_Client_Client($config);

// Detect your device
$client->detectDevice();

// Use the capabilities
if ($client->getDeviceCapability('is_tablet')) {
    echo "This is a tablet";
}
```

```
}  
?>
```



Remember that when using the WURFL Cloud Service you don't need to download or update any WURFL repository on your server. When you install the Cloud Client, you will always be accessing the latest database.

DeviceAtlas

In February 2008 (many years after WURFL was created), the dotMobi company, which owns the *.mobi* top-level domain, launched its own device database that is similar in many ways to WURFL.

DeviceAtlas (<http://deviceatlas.com>) is a commercial product that has partnerships with many data providers.

The main features are:

- Monthly, weekly, daily, or constant updates to the database, depending on your license
- A data explorer to browse the database from the Web
- JSON data format support
- APIs for PHP, Java, .NET, Python, and Ruby
- Apache server module (with the enterprise license)
- Cloud-based service available

At the time of this writing, the service is available as an Enterprise plan for standalone installation with no public prices and as a cloud-based service with two plans available:

Standard (\$40/month)

1 million detections per month, single applications, weekly updates, standard set of capabilities

Premium (\$399/month)

5 million detections per month, 50 applications, daily updates, premium set of capabilities



At the time of this writing, DeviceAtlas has no free option (standalone or cloud-based), unlike WURFL. You can, however, request a free evaluation license for testing purposes.

The Enterprise API is available for Java, PHP, .NET, Python, and Ruby. You'll need to download the JSON repository yourself, at whatever frequency your license allows. The cloud-based API is available for Java, PHP, .NET, and Python.

Properties. The data available in DeviceAtlas is segmented into categories. In the cloud-based services, some properties are available only with the premium plan.

The most important properties in each category are:

Device name

vendor, model

Hardware

displayHeight, displayWidth, mobileDevice, touchScreen, displayPpi, isEReader, isTablet, isMobilePhone, isGamesConsole, isTV, isMediaPlayer

Environment

developerPlatform, developerPlatformVersion, osAndroid, osLinux, osOsx, osProprietary, osRim, osSymbian, osWindowsPhone, osVersion

Web browser

markup.xhtmlMp10, memoryLimitMarkup, uriSchemeSms, uriSchemeSmsTo, uriSchemeSmsTel, vCardDownload, usableDisplayWidth, usableDisplayHeight, flashCapable

JavaScript

js.supportsBasicJavaScript, js.xhr, js.json

Network protocols

EDGE, GPRS, HSDPA, UMTS

HTML5

html.audio, html.canvas, html.svg, html.video, js.applicationcache, js.geolocation, js.touchevents

AudioPlayer

aac, amr, mp3

Streaming

stream.3gp.aac.lc, stream.3gp.h263, stream.3gp.h264.level1, stream.mp4.aac.lc

VideoPlayer

3gp.h263, 3gp.h264.level1, mp4.aac.lc, wmv

You can browse all the data available with your license at DeviceAtlas (<http://deviceatlas.com/explorer>).



Remember that if you have a developer account (the free evaluation license) your database file will not be updated if you download it again, and if you have a basic license with monthly downloads a new file will not be available until 30 days from when you downloaded the previous version.

Cloud API. For demonstration purposes, we are going to use the PHP Cloud API from DeviceAtlas.

Inside the PHP Cloud API you will find just a *DeviceAtlasCloud/Client.php* file that you need to add to your project.

Then, your PHP will look like this:

```
<?php
include './DeviceAtlasCloud/Client.php';

// Get the data - this automatically uses the browser/device User-Agent
// If TRUE, then a test device will be used to query DeviceAtlas Cloud
$test_mode = true;

$da_data = DeviceAtlasCloudClient::getDeviceData($test_mode);

if (isset($da_data['properties']['isTablet'])) {
    echo "This is a tablet";
}

?>
```

If you want a cache implementation, you'll need to do it yourself.



Microsoft used to have an open source mobile browser database (<http://mbf.codeplex.com>) for ASP.NET. This project was abandoned; now you can use the *.browser* file to detect a device library such as WURFL, or see other ideas at ASP.NET (<http://www.asp.net/mobile>). ASP.NET MVC is recommended instead of Web Forms to create mobile websites.

DetectRight

DetectRight is a detection engine, device database, analytics engine, and API/SDK with both service-based and dedicated server options. It offers a standalone API, a cloud service, and a local cloud service for enterprise applications.

Free nonprofit, community, and developer licenses are available, as well as commercial enterprise licenses. DetectRight features include SOAP and REST access, unique custom identification, country-level geolocation, and profiles in WURFL, DeviceAtlas, UAProf,

DetectRight, and Java ME Polish-compatible formats. Its database includes over 20,000 devices at the time of this writing.



If you want easy and quick mobile detection, at Detect Mobile Browsers (<http://www.detectmobilebrowsers.mobi>) you will find a little piece of PHP code that allows you to determine whether the user is using a mobile browser, without any repository, database, or service call.

If you register at DetectRight (<http://www.detectright.com>), you will receive via email a key that enables you to access a variety of PHP, Java, and SOAP samples and APIs.



Remember that when using cloud-based solutions, your mobile website's performance will depend on the reliability of the third-party server to which you are connecting.

You can download the PHP or Java API for easy usage on those platforms.

In PHP, once you've downloaded the API, you can use the service as shown in the following sample:

```
include_once("detectRight.php");

DetectRight::$druser = '<detectright.com username>';
DetectRight::$drpassword = '<detectright.com password>';
// possible values: DR, WURFL, W3C, UAProfile, J2MEPolish
DetectRight::$defaultSchema = 'DR';

$profile=DR_Customer::deduceCustomer($_SERVER);
$value = $profile['<property>'];
```

The API can also be used to download lists of manufacturers and devices, and to request individual profiles by manufacturer/model name.

Movila DetectFree

Movila Detection (<http://www.moviladetection.com>) is a server-side Java solution to detect in 500 microseconds which device is using an embedded repository. It also works as a tool for URL rewrites. However, Movila's most-used feature is the free service called DetectFree.

DetectFree is a free light version of the service available for PHP and JavaScript (and for any other platform that sends HTTP requests) that allows you to detect whether the connecting device is a mobile device. You can find samples and documentation at DetectFree's website (<http://www.moviladetection.com/detectfree>). Just to illustrate how easy it is to use, the following sample is a JavaScript detection mechanism:

```
<script src="http://detectfree.mobiladetection.com/detectfree.js"
  type="text/javascript"></script>

<script type="text/javascript">
if (is_mobile) {
  alert("This is a mobile device");
}
</script>
```

51Degrees.mobi

51Degrees.mobi (<http://51degrees.mobi>) offers free and premium tools for mobile device detection, providing open source APIs for PHP, Java, C, and .NET. The device database itself is delivered as an XML file, with two options:

Lite data file

A free version offering basic user interface optimization properties, optimized to just separate mobile from nonmobile traffic.

Premium data file

The full commercial database of phones, tablets, ebook readers, games consoles, and more, with a big list of properties available. The price starts at \$360 for an annual subscription.

51Degrees.mobi's data used to rely on the WURFL database, but a different scheme and vocabulary were developed after ScientiaMobile restricted the WURFL license.

OpenDDR

OpenDDR (<http://www.openddr.org>) is a recent project whose stated goal is to offer a community-based and open source device database. At the time of this writing, APIs are available only for Java and .NET, but the device format used is based on the W3C DDR format, which is as an open format that can be used from any language.

The APIs and the database are free to use. The device library started with the last WURFL repository version that was completely free, before ScientiaMobile changed the license; some of the OpenDDR vocabulary is also based on WURFL.

New versions of the database are being distributed monthly, and anyone can contribute to it. Compared to other libraries, the most important criticism is the lack of some capabilities; other drawbacks include its dependence on WURFL, a not-so-easy-to-use API, and the contributors' anonymity, which may have an impact on support and maintenance.



Apache DeviceMap (<http://incubator.apache.org/devicemap>) is an incubating project whose aim is to create a data repository for mobile device information. At the time of this writing, no product has been released, but OpenDDR has announced its forthcoming incorporation into DeviceMap; in the future we may see both projects merging.

Images and Media

It's often said that a picture is worth a thousand words. This is true in the mobile web, too. However, we need to find a balance with regard to the number of images and media elements in a document. Every image or resource adds to the network traffic, number of requests, and load time, unless we use best practices and techniques to optimize performance.

Let's take a look at what options we have for images and other media elements, such as video and audio playing.

Images

When we talk about images, your first thought is probably the `img` element. While this is one of the most-used solutions for images on the Web, it's not the only one available in mobile HTML5. With HTML5 we can render images with Scalable Vector Graphics (SVG), the canvas element, and Cascading Style Sheets (CSS).



With CSS3 we can replace images for a lot of use cases, such as gradients, rounded corners, shadows, and other effects. If you can do it with CSS, do it and don't use images.

Image Formats

There's good news here. Almost every mobile browser understands standard static web image formats: GIF, JPEG, and PNG. On tablets and modern smartphones, all of these formats will work without any major issues. That said, there are some differences with regard to indexing and alpha transparency on devices shipped before 2011.

Some mobile browsers also support SVG as a replacement for a bitmap image, as in the `img` element or `background-image` in CSS.

Image Size and Memory Consumption

When you load an image in a web application, the original file size matters only in the data transfer. Once the file is on the browser, it is decoded in memory and it is always treated as a bitmap. Therefore, the size in memory can be calculated as $\text{width} \times \text{height} \times \text{bits per pixel}$. We should be careful, therefore, about loading big image files even if they are small in terms of bytes. Just to give an example, a full-screen image on a third-generation iPad will typically use around 12 MB of RAM.

Using more RAM will impact website performance when loading, scrolling, and returning from a frozen state (for example, when changing tabs).

A new format that is compatible with some mobile browsers is *WebP* (read it as “wep-py”). WebP is an open format created by Google, based on the VP8 video codec; it is intended to replace the JPEG and PNG formats as it has a better compression algorithm for both lossy and lossless images. For example, an image of the same image quality will have a file size in WebP that is, on average, 25%–45% smaller than in JPEG. WebP is also smaller for lossless images (on average, a WebP image is 26% smaller than the same image in PNG format). Therefore, it’s a good candidate for the mobile web, as it is likely to enhance performance (less traffic, less time).



To find out more about the WebP format and conversion tools, you can check out the Google Developers WebP page (<https://developers.google.com/speed/webp>).

The problem with WebP files is compatibility: at the time of this writing, only the Android browser, Opera (both Mobile and Mini), and the UC browser have complete support. Google Chrome for Android supports it, but there’s a bug in version 18 that changes the color palette, making it useless.

Animation formats

For animation, the standard in mobile web development is the *animated GIF*. As Flash support isn’t included in many browsers (as you’ll see later in this chapter), and even when it is included it can be slow, banners and animations will be most widely compatible using this classic format, until SVG or other solutions become more widely supported.

Animated GIFs were having problems before 2011, such as on Android 2.0, but today support is good enough to use them without any problem.

APNG (Animated PNG) is an unofficial standard for using PNGs for animations. The draft specification can be found at the Mozilla wiki (https://wiki.mozilla.org/APNG_Specification). At the time of this writing, only Opera Mobile and Firefox since version 14 support APNG on mobile websites. The polyfill *APNG-canvas* should make APNG work with Safari on iOS, the Android browser, and Chrome. Browsers that don't support APNG will just render the first frame of the animation. Modernizr has an extension attribute, *apng*, that will give us current compatibility information.



APNG-canvas is a polyfill that is compatible with mobile browsers that use the Canvas API to run the animation. You can download it and check out a demo at Davidmz's GitHub (<https://github.com/davidmz/apng-canvas>).

The advantage of APNG over animated GIFs is the ability to use alpha channels and 32-bit images. A quick way to create APNG files is using APNG Edit, a free Firefox for desktop plug-in available from Mozilla's Add-Ons (<http://bit.ly/16xxyZE>).



An SVG element can also be animated, and it can be packaged as an *.svg* file or as a full URL that we can load as an *iframe* with all the contents inside.

Later in this book we will also cover CSS3 transitions and CSS3 animations.



Older devices had support for XHTML+SMIL animation format (<http://www.w3.org/TR/XHTMLplusSMIL>). It is an obsolete standard today.

3D formats

As some mobile devices—such as the Evo 3D and LG Optimus 3D—have 3D screens, as we saw in Chapter 1, we can use some 3D image formats. However, having a 3D screen doesn't mean that those devices browse the Web in 3D. Therefore, usually we'll only use these formats to deliver content that the user can open, if desired, in a 3D image viewer.

The most often used format is the JPS (JPEG Stereoscopic) format, which is basically just two JPEG images side by side (one for each eye). If you use the JPEG extension, viewers will just see two images, one beside the other. Special 3D software will understand it better if the JPS extension is used; this tells it to load the image in the right way.



Always compress your images as much as possible to reduce bandwidth consumption and improve performance. Cloud-based browsers will compress your images automatically, sometimes with quality loss. In these browsers it's the user who decides on the image quality, not the developer.

Inline images

A *data URL* (<http://www.ietf.org/rfc/rfc2397>) or *data URI* is a mechanism for defining a URL with embedded content (such as an inline image), defined in 1998. For example, we can define an `img` tag with the image itself inside it, without using an external file. This can be done using a base64 encoding of the image file—basically, storing the binary file as a set of visible ASCII characters in a string. This is great for small images, icons, backgrounds, separators, and anything else that doesn't merit a new request to the server. Where is the catch? No catch! All modern mobile browsers are compatible with this feature.



The size of an image (or any other binary file) will increase by about 30% when it's converted to a base64 string for a data URI, but its size will be reduced again if we are serving the document using GZIP from the server. Therefore, at the end it will be the same size or even smaller, and it won't require a new request (with all the overhead that involves).

The best part about data URIs is that they can be used in a CSS file, with caching and multipage support, or they can be stored in any string-based storage. We will cover that later in this book.

To convert an image file to a base64 string representation, we can use any online converter or command-line utility. There are free and online alternatives at Web Utils (<http://bit.ly/13XAqza>) and Base64 (<http://www.base64encode.org>).

PHP Base64 Conversion

Many web server platforms offer base64 conversion. For example, PHP offers a `base64_encode()` function for this purpose. To generate the code based on a real file on your server, use something similar to `base64_encode(file_get_contents($path))`. You'll need to add error support and insert the result in an `img` tag.

The syntax is `data:[MIME-Type]([;base64]),data`. The *data* can contain spaces and newlines for readability purposes, but some browsers won't render it properly. It's better to maintain it in one line.

For example, the O'Reilly logo (original PNG file 75 pixels wide) attached as a data URI image looks like this:

```

```

I know what you're thinking: "This is awful, didn't you tell me to create clean HTML?" Maybe you're right. However, we can automate the encoding, or use it in many other ways, such as adding it in CSS or loading multiple images at the same time using Ajax techniques and a JSON file.



You should deliver data URI images only to known compatible browsers. In Modernizr, you can check the `datauri` attribute, and in WURFL the capability is `image_inlining`.

In CSS this image could be a background-image, like:

```
#logo {
  width: 70px; height: 17px;
  background-image: url('data:image/png;base64,
iVBORw0KGgoAAAANSUHEUgAAAEYAAARBAMAAACSi8f4AAAAA3NCSVQICAjb4U/gAAAAAGF
BMVEX/////////8AAACpqnMzMxmZmaHhoQ/Pz9kt3AEAAAACHRSTLMA/////////9XKVDIA
AAAJcEhZcwAACxIAAAsSAdLdfvWAAAACdEVYdFNvZnR3YXJlAEFkb2JlIEZpcmcV3b3Jrcy
BDUzQGstOgAAAFnRFWHRDcmVhdGlvbiBUaW1lADEyLzExLzA5uegApgAAQNJREFUKJGV
kUFTwyAQhfMXXiH1LA3hDMTEsVDPidW7WnMvkxn/vo+MsamX6s7wg0y3029JgetR/I2Rdy
8B8HIcAyhj8PLIr0dsuYLDf8i8KWWC/BRKaVCUTaJmkhcDuNOHLVCaJg6VfJY6JivqGJHE
jn00q3tpsGcdF0KuuGdx2+fsmSL3m2r2k2gGg30i434xSMLmht0YbR+EFLAU71dMMW89zzW
cy2W61eF6YssK5j3vLIi81Lj0vzOKHaXCuS5z8334ybgC2bkdlUK6ZeMvVtDMMOM0IL3fmQ
skbD08LA8YHDzGDw9Nyn7HzIuv1hsH/PltCQi9770MmsXFAGf/z3q/EFatLL/IFsBmgAAA
AASUVORKSCYII='') top left no-repeat;
}
```

The best part about using data URI images in CSS is that we can use them in more than one document without downloading the image content again (by caching the same CSS), and we still have clean XHTML.



With HTML5 features, you can save in a database or in a JSON object a list of base64 image files that will be cached for the user on the device for future usage in URLs. See Chapter 13 for more information.

Format compatibility

Table 10-1 explores image features and the browsers that support them. Remember that static GIF, PNG, and JPEG formats are supported on all platforms.

Table 10-1. Image format compatibility table

Browser/platform	WebP	Animated GIF	APNG	APNG with polyfill	SVG as an image format	Data URI
Safari on iOS	No	Yes	No	Yes	Yes	Yes
Android browser	No up to 3.2 Yes from 4.0	No up to 2.1 Yes from 2.2	No	Yes	No up to 2.3 Yes from 3.0	Yes
Chrome for Android	Yes (buggy on 18)	Yes	No	Yes	Yes	Yes
Nokia Browser for Symbian	No	Yes	No	No	No	Yes
Nokia Xpress browser	No	No	No	No	No	Yes
BlackBerry browser	No	Yes	No	Yes	Yes	Yes
Internet Explorer	No	Yes	No	No up to 9 Yes from 10	No up to 7.x Yes from 9	Yes
Firefox Mobile	No	Yes	Yes	Yes	Yes	Yes
Opera Mobile	Yes	Yes	Yes	Yes	Yes	Yes
Opera Mini	Yes	No	No	No	Yes	Yes
Amazon Silk	No	Yes	No	Yes	No	Yes
UC browser	Yes	Yes	No	No	No	Yes

Using the `img` Element

The `img` element should be used only for semantic images, or images that deserve to be images. That means, don't use the `img` tag for:

- Buttons
- Icons for links or menus
- Backgrounds
- Visual separators
- Titles

That doesn't mean that you shouldn't use images for any of those purposes—just don't use the `img` tag. That tag is semantically correct for images that the user understands as images in their own right, not for visual aids. Most users don't consider an arrow icon to be an image; it's just a button, or a link. We will follow the same rule.

The mandatory attributes for an `img` tag are `src`, `width`, `height`, and `alt`. It is very important to define the width and height of every image in a mobile document. This

will reduce the initial rendering time, because the mobile browser won't need to wait for the image to load to know how much space it will take up and how to draw the rest of the content.

The alternative text (`alt` attribute) is also mandatory, because the user can disable images or they can be very slow to load, and the document must work without them. The `alt` text should provide enough information for the user to understand what is missing.



Image maps should be avoided on the mobile web. Focus-navigation devices don't have image map support, and while cursor-navigation devices may support them, usability is a problem for the users. They can even be problematic on touch-navigation devices, because of the finger size. Only use image maps when targeting compatible devices, and if those devices support touch navigation, use large areas for every link.

Out of all of these complications and possibilities, some guidelines emerge:

- Use images in HTML only for logos, photos, and maps.
- Compress the images with normal web image methods.
- Define the width, height, and alternative text for every image.
- Use data URIs for small images whenever possible.
- Leave icons, buttons, backgrounds, and visual alert images for CSS.
- Open your mind to the usage of Emoji and pictograms with known compatible devices.
- Avoid the usage of image maps.
- Analyze the use of canvas or SVG for compatible devices and for some graphic types. These technologies will be covered later in this chapter.

Responsive Images

When developing for the mobile web, we need to adapt to different contexts, and when using Responsive Web Design we need a way to have adaptive images, or images that are available in different sizes for different contexts. The ability of a web app to adapt images involves a technique called *responsive images*.



Matt Wilcox wrote a good article on problems with responsive images and possible solutions, titled "Responsive images: what's the problem and how do we fix it?" (<http://bit.ly/XyWte9>)

When using Responsive Web Design techniques and background images, we can make it work with background-size and background-position to change the image's clip based on the current available screen size or orientation. In Figure 10-1, we can see how Food Sense (<http://foodsense.is>) applies this idea—same background image, different size and clip.



Figure 10-1. The Food Sense website changes the size and position of the background image to create different masks of the same image and support different layouts and orientations in the same HTML file.

A different solution that can be applied in different scenarios is to have a really big image on the server that will be automatically resized to suit the current context—mixed with

some JavaScript or cookies, this can help us deliver the right image for every context. We can use a client-side framework that will leave a cookie so the server can determine the current necessary image dimensions, or we can use any server-side library, like WURFL to determine the correct image sizes (as we saw in Chapter 9). One already created solution (involving server-side code) is available from Adaptive Images (<http://adaptive-images.com>).

Sencha.io Src

Sencha (<http://sencha.com>), the company behind the Sencha Touch framework and many other solutions for web developers, offers a free service for creating responsive images in one second: the Src service. The documentation is available in Sencha docs (<http://docs.sencha.io/current/index.html#!/guide/src>); it's really easy to use and we don't need to install anything.

Let's say we have a logo that we want to be 100% the width of the device's screen, such as:

```

```



Sencha.io Src always needs an absolute URL for the source definition, as it doesn't know where to get the file if we use a relative URL.

The problem with this approach is, what size should we use for the file? If we deliver a small file it will be resized and lead to poor visual results; if we deliver a large file we are wasting bandwidth and valuable loading time. Instead of having different sizes or providing a server-side algorithm to resize the file, we can just add `http://src.sencha.io/` as a prefix for the image URL:

```

```

That's all? Yes. The image request will be done to the free Src service, which will take the image from our server and resize it automatically to the current device's screen width. Therefore, the image file that is transferred from the server to the browser will be the most suitable one.

The service offers us different options to customize how it delivers the file. For example, if we don't want to use 100% of the device's screen as a limit, we can add a different percentage using `x<percentage>/`, as in:

```

```

We can also make some other complex calculations, such as a percentage of the screen minus a fixed margin value, as in `35% - 20px`:

```

```

And we can force the image to take landscape values:

```

```

Another great feature is the data URI conversion (discussed earlier in this chapter), which we can call using data:

```
http://src.sencha.io/data/http://mobilehtml5.org/img/logo.png
```

We can use a JSONP technique to get a callback when the data URI is available, creating a callback function such as `dataReady` and defining its name inside the script we are injecting:

```
<script src="http://src.sencha.io/data.dataReady/http://  
mobilehtml5.org/img/logo.png">  
</script>  
<script>  
function dataReady(dataURI) {  
    // Do something with the data URI representation of the image  
}  
</script>
```

The API offers lots of other features that we can check on the Sencha website. As an experimental service, Sencha offers a JavaScript file that we can also add that will detect orientation changes and resize the image accordingly.



Firefox for mobile devices supports the `image-rendering` CSS attribute, which accepts different values (such as `optimizeSpeed`) that change how a bitmap image is scaled when zooming in or on high-resolution devices. More information about this property can be found at the Mozilla Developer Network (<http://mzl.la/Ydew4b>).

Local Pictograms

The Japanese carriers have created a de facto standard for using small icons in HTML without really using images and requests. The images (called *Emoji*, a Japanese word combining picture and letter) are based on a list of dozens of icons available to use, with the real rendering done by the browser. For example, say you would like to insert a heart icon. Every compatible browser will display a heart icon, but you might not know the exact image that the browser will use. Today, NTT i-mode services include *Basic Pictograms* (176) compatible with all devices and *Expansion Pictograms* (76) added in HTML 4.0. Other Japanese carriers (Au and Softbank) have their own pictograms (and their own usage mechanisms), so if you need to cover all Japanese carriers there are a lot of conversion tables between codes.

To display these pictograms, you can just embed the binary code into your HTML (saved as a Shift-JIS file, not UTF-8) or define them as Unicode standard characters using `香`, where 9999 should be replaced with the pictogram number. The list of possible pictograms can be found at www.mobilexweb.com/go/pictograms.

This is a very underused feature in the occidental world. Even Safari on iOS supports these icons, although I have not seen too many websites using them. iOS has supported an Emoji keyboard since version 2.2 (though only for devices sold in Japan), and the browser allows Emoji pictograms for users worldwide.

The list of iPhone Emoji is available at Wikipedia (<http://en.wikipedia.org/wiki/Emoji>) and is best browsed with an iOS device. The list is long—there are more than 450 Emoji—and the icons have great designs. Moreover, remember, they are images that don't use network resources! For example, for the iPhone we can show a message with a smiley at the end, as shown in Figure 10-2, with the following code:

```
<p>Thanks for your message! &#xe415;</p>
```



Figure 10-2. Safari on iOS shows the Emoji icon, but other older browsers show only a rectangle indicating a noncompatible character.

There is also a polyfill available that will use high-resolution PNG files using the same iOS Emoji codes. You can see the full list and download the polyfill at code.kwint.in/emoji/.



Remember, iPhone Emoji work all over the world, not just in the Japanese market. You can detect compatibility dynamically using the Modernizr extension property `emoji` or the WURFL capability `emoji`.

ISO pictograms

In the last few years, Unicode 6.0 (<http://www.unicode.org/charts>) has standardized some emoticons and other characters from Apple and Android proposals under the name *ISO pictograms*. At the time of this writing, Android (since 4.1), Safari on iOS 6.0, and Windows Phone (since 7.5) support ISO pictograms, either totally or partially. ISO pictograms use the same syntax as Emoji characters on iOS, using HTML-encoded Unicode characters.

A list of emoticons and other symbols that we can use today (and their compatibility with modern browsers) can be found at MobiForge (<http://bit.ly/WfjeSu>).

For example, to show a star symbol we can just use:

```
<p>A star symbol: &#127775;
```

OMA pictograms

The Open Mobile Alliance (OMA) standardized pictograms in XHTML MP using an object tag like this:

```
<object data="pict:///core/arrow/right" />
```

The standard supports alternative content. That is, if the pictogram is not available, you can add a child to the object with an alternative. The alternative can be another pictogram or a classic image:

```
<object data="pict:///time/season/summer">
  <object data="pict:///weather/sunny">
    
  </object>
</object>
```

The pictogram sets are not standardized between browsers, and that's why today they are not widely used in mobile websites.



A good resource for Emoji and pictograms is the Google site for Emoji symbols (<http://bit.ly/XyXP8G>).

Dealing with Multiple Screen Densities

As discussed earlier in this book, in the mobile world we have to deal with screens with different pixel densities, or DPI (dots per inch), and DPI always has a direct relationship with images. If you remember from Chapter 2, we have devices with low, medium, high, and ultra-high density. If we don't take this into account, we may get undesirable results.

Recall also that when we use HTML or CSS px dimension values, they are expressed in CSS pixels, or resolution-independent pixels. And we already know from Chapter 7 that our dimensions will be calculated based on the current viewport and zoom level.

Therefore, when we say:

```

```

we are not saying “always render this logo at 200×80 pixels.”

The same is true when we use:

```
#logo {  
  width: 200px; height: 80px;  
  background-image: url(logo.png);  
}
```

We are saying lot of things at the same time here, such as (with a viewport using device-width and no scaling):

- On a medium-density screen (such as the iPhone 3GS), render the image at 200×80 pixels.
- On some other medium-density screens (such as the Nexus 7 tablet, with a pixel ratio of 1.325), render the image at 265×106 pixels.
- On a high-density screen (such as the Samsung Galaxy SII, with a pixel ratio of 1.5), render the image at 300×120 pixels.
- On an ultra-high-density screen (such as the iPhone 5, with a pixel ratio of 2), render the image at 400×160 pixels.

Using Vector-Based Solutions

The first option that does not cause image-definition problems or require delivering different files is to use a vector-based solution. The possible implementations are:

- Use SVG for any nonbitmap content, such as logos or icons. SVG is scalable, so every device will get the right resolution by default. Always provide an alternative for noncompatible browsers, such as version 2.x of the Android browser.
- Use CSS effects. Everything that is rendered by CSS will have the right resolution by default. That includes gradients, shadows, rounded corners, reflections, and other effects. We will cover these in Chapter 11.
- Use custom fonts. Instead of using fonts in images, you can use custom fonts. Fonts are scalable by default and they will have the right resolution for the current device. You can even create your own fonts and use them for iconography or other vector-based assets in your app. We will cover how to use your own fonts in Chapter 11.



IcoMoon (<http://icomoon.io>) is a free online service that creates an icon gallery as a custom font. It includes lots of icons that are ready to use, and you can upload your own scalable icons to the gallery.

Custom Fonts Versus SVG

When creating a vector-based icon gallery, SVG and custom fonts are two possible solutions. In most situations creating custom fonts will result in much smaller files, but there are also some disadvantages, such as the icons being monochrome only.

SVG, on the other hand, can support multiple colors, more complex paths, and gradients; it can also be manipulated from the DOM API in JavaScript, and SVG files are based on XML, so they are easier to edit than custom font files.

Providing One Single Image

A simple approach is to provide a single file (say, 200×80 pixels, 400×160 pixels, or something in between) that will be rendered at different sizes on different devices. The problems with that solution are:

- If we provide an image smaller than the current density (for example, a 100×100 image on a device with a pixel ratio of 2 that is expecting a 200×200 image), the browser will resize the image, and that means we will lose quality. Even if the result is not so bad and there are no visual glitches or problems, the user may realize that the image does not have “good resolution” and is not as sharp as it could be.
- If we provide an image larger than the current density, the browser will again resize the image. Usually resizing down doesn’t cause problems in terms of visual quality. The problem is that we are delivering a big file (from 2 to 4 times bigger than necessary), meaning more bandwidth, a longer download time, and more memory consumption. And usually devices with lower density are older or cheaper, meaning they have less power and less memory.

Using `img` elements

If we decide to provide an `img` image with a resolution higher than the medium standard one, it’s important to explicitly define the width, height, or both. If not, we may have a problem. Let’s say we use:

```

```

and we expect the logo to be 200 CSS pixels wide. If we provide a 400-pixel-wide image (ultra-high density), the browser will use 400px as the width value, so it will render at

double the size. Even worse, on ultra-high-density screens it will be rendered as 800 device pixels wide (400 multiplied by the pixel ratio of 2).



When providing an `img` image with a resolution higher than medium, you must explicitly define the dimensions, in CSS pixels.

Using background images

The problem is worse with CSS background images. Going back to our example:

```
#logo {  
  width: 200px; height: 80px;  
  background-image: url(logo.png);  
}
```

In this case we are defining a 200-CSS-pixels-wide box with the logo image. If we define the image as double the size for ultra-high-density devices, we will get the wrong effect, as seen on the left in Figure 10-3. The background image size will be multiplied by the pixel ratio automatically. To solve the problem, we can use the new CSS3 `background-size` property. That property allows us to define the size in CSS pixels or as a percentage of the background inside the box.

If we want to use percentages in `background-size`, for high-resolution images we should always use 100% to match the box's size. We can also use specific values (width first, then height) so we get the desired result, as seen in the image on the right in Figure 10-3:

```
#logo {  
  width: 200px; height: 80px;  
  background-image: url(logo.png);  
  background-size: 200px 80px;  
}
```



Figure 10-3. Left: high-resolution image with the `background-size` equal to the image's size. Right: same image with a `background-size` equal to the CSS element's size.

Providing Image Alternatives

As the Web evolves and changes, new features, techniques, and solutions appear in the browsers. That means if we already have variants of the same image for different resolutions, we can use some new techniques to deliver them. These techniques include:

- JavaScript and the `devicePixelRatio` property
- CSS media queries
- Image sets

As we saw in Table 2-2, there are different devices out there with different pixel density ratios. Providing a different image for every possible pixel ratio seems an impossible mission, as new devices are constantly appearing on the market.



As we may need images with more density in the future, it's a good idea to always store vector-based versions of all our logos, icons, and so on, and high-resolution versions of all our bitmap images (around 300–400 DPI).

The best solution is to define ranges. For example:

- Version 1, from 0.75 to 1.25: your original CSS pixel size
- Version 2, from 1.25 to 1.75: your size multiplied by 1.5
- Version 3, from 1.76 to infinite: your size multiplied by 2

Some resizing will still occur, but the results should not be so problematic. To name your files, you can have different folders, such as *images/*, *images-1.5x/*, *images-2x/*, or use the same filename with a suffix, such as *logo.png*, *logo-2x.png*, *logo.2x.png*.



In native iOS development, the convention is to use the *@2x* suffix for high-resolution images. In HTML5, we should avoid using the *@* sign (*@*) inside filenames to avoid URL problems.

Using `devicePixelRatio`

Some browsers expose the current device pixel ratio as a JavaScript variable of the window context, called `devicePixelRatio`. You should always provide a default value of 1 if the attribute is not defined (the assumption being that it is an older device with a medium-density screen):

```
var pixelRatio = window.devicePixelRatio || 1;
if (pixelRatio <= 1.25) {
```

```
// Group #1, low and medium resolution
} else if ((pixelRatio > 1.25) and (pixelRatio <= 1.75)) {
    // Group #2, medium and high resolution
} else {
    // Group #3, ultra-high resolution, usually 2, 2.25
}
```



Because iOS devices often have a pixel density ratio of 2, it's a common technique to search for this exact ratio. That is not a good practice, as there are devices with higher density than iOS devices that deserve high-resolution images, but they receive the low-resolution versions just because their pixel density ratios are greater than 2. Always use greater than or less than when specifying resolutions inside conditionals.

When you detect every group, you can then define the corresponding image URL using the `src` attribute or CSS styles. The main problem with this technique is that the browser will not start downloading and rendering any image until your JavaScript code is executed.

Another solution, used by Apple on its home page, is to always load the low-resolution version first: the JavaScript code will replace it as soon as possible. The main problem with this approach is that in many cases the browser ends up downloading two versions of the image(s), wasting time and traffic.

Using media queries

We will cover media queries in depth in Chapter 11. For now, suffice it to say that it is a CSS3 feature allowing us to define different styles for different conditions. Most browsers with different densities support an extension to the media queries standard allowing us to query about the pixel ratio in the CSS itself.

The browsers call the pixel ratio extension `device-pixel-ratio`, and it's prefixed. To make a “greater than” condition, we can use `min-device-pixel-ratio` instead. To further complicate our lives, the WebKit prefix version is `-webkit-min-device-pixel-ratio` and the Mozilla version is `min--moz-device-pixel-ratio` (with a double `-`).

Because the CSS standard already had a similar property for the pixel ratio (`resolution`) but using `dpi` as the unit, the W3C group decided in mid-2012 to define a new unit so it wouldn't need to formalize the nonstandard `device-pixel-ratio`. The new unit is `dppx` (dots per pixel), and it's equal to the pixel ratio value. Therefore, for future compatibility we can start adding the standard `resolution` version.



Opera with Presto engine supports the prefixed pixel ratio extension but accepts fractions (the relation between resolutions) instead of float values: so, 1.5 becomes $3/2$ and 2 becomes $2/1$.

To give an example, we can make the following groups using CSS:

```
/* Definitions for all resolutions; dimensions will stay the same in
   CSS pixels */
#logo {
    width: 200px; height: 80px;
    background-size: 200px 80px;
}

/* We define the default value out of any media query */
#logo {
    background-image: url(logo.png);
}

/* We use different declarations for all the possible prefixes */
@media screen and (min--moz-device-pixel-ratio: 1.5),
    only screen and (-o-min-device-pixel-ratio: 3/2),
    only screen and (-webkit-min-device-pixel-ratio: 1.5),
    only screen and (min-device-pixel-ratio: 1.5),
    only screen and (min-resolution: 1.5dppx) {

    /* This definition will replace the default value when the pixel
       ratio is at least 1.5 */
    #logo {
        background-image: url(logo_1.5x.png);
    }
}

/* We use different declarations for all the possible prefixes */
@media screen and (min--moz-device-pixel-ratio: 2),
    only screen and (-o-min-device-pixel-ratio: 2/1),
    only screen and (-webkit-min-device-pixel-ratio: 2),
    only screen and (min-device-pixel-ratio: 2) ,
    only screen and (min-resolution: 2dppx) {

    /* This definition will replace the default value when the pixel
       ratio is at least 2 */
    #logo {
        background-image: url(logo_2x.png);
    }
}
```



Before the device-pixel-ratio extension to the media queries standard was created, the spec included the resolution conditional, which accepts DPI values. We can convert from pixel ratio to DPI easily; the problem is that not every browser supports the resolution conditional, as we might expect. Therefore, some devices will honor 96dpi, no matter what the real resolution is. The new unit dppx will solve this problem in the near future.

You may be thinking about changing the `background-size` for every density definition—for example, using `background-size: 50%` on devices with a pixel ratio of 2 because the image size is double that of the original. However, you need to remember that when you give `background-size` a percentage value, that value applies to the image container—that is, the HTML element. And in our case, the HTML element is always the same size in CSS pixels, no matter what resolution the device has. Therefore, the `background-size` should not change from its original value.



Instead of specifying dimensions in the `background-size` property, we can use the constant `cover` that will just fit the image into its container.

Using image sets

Safari on iOS 6 has added a new nonstandard feature to support multiresolution images: the `image-set()` CSS function. As an experimental feature, it's prefixed, so the final name is `-webkit-image-set()`. At the time of this writing, only Safari on iOS 6 supports it.

`image-set()` is a CSS image function, meaning that we can use it any time we can use an image in CSS, such as in `background-image`. It accepts a set of comma-separated declarations, each consisting of an image and a resolution.



Image sets have been added to the CSS4 standard discussion as a Resolution Negotiation feature (<http://bit.ly/Ydf6Pe>).

As with any other CSS property, we can make a fallback by defining a standard value first. This should become clearer with an example:

```
#logo {  
  width: 200px; height: 80px;  
  background-size: 200px 80px;  
  background-image: url('logo.png'); // Standard fallback  
  background-image: -webkit-image-set(url('logo.png') 1x,  
    url('logo-2x.png') 2x);  
}
```

In this case, in one declaration we are defining two images, one for a pixel ratio of 1 and the other for a pixel ratio of 2. Remember that iOS supports only devices with pixel ratios of 1 and 2 at the time of this writing. If other browsers, such as Chrome for Android, begin supporting this feature in the future, it's probable that they may use other values, such as 1.5x.

The main advantage of image sets over CSS media queries is the simplicity of the syntax and the fact that we don't need to search all over our stylesheets to find all the possible values for the same image on devices with different resolutions.

The picture element

The main problem with using CSS media queries or image sets is that we are always talking about CSS background images. From an HTML perspective, there is no way to do the same thing without adding JavaScript code. In recognition of this, the W3C has started a new discussion group—the Responsive Images Community Group (<http://bit.ly/ZPtUaL>)—that is focused on creating one new element for the future of HTML5: the picture element.

The picture element draws upon a mix of media query and image set techniques, but from an HTML point of view, to replace the `img` element. At the time of this writing no browsers support this new element, but some probably will soon.



Scott Jehl (<http://scottjehl.com>) has created an open source polyfill for the picture element that validates against HTML5, using divs with a very similar syntax to the W3C's proposed picture element. It's available at Scott Jehl's GitHub (<https://github.com/scottjehl/picturefill>).

The picture element can have different source elements, providing different images by defining different media attributes and one `src` or `srcset` definition (following the image set syntax). Let's look at a sample:

```
<picture width="500" height="500">
  <source media="(min-width: 50em)" src="large.jpg"
    alt="Single image by media query">
  <source media="(min-width: 20em)"
    srcset="medium.jpg 1x, medium_2x.jpg 2x"
    alt="Multiple image set">
  
</picture>
```

Any browser that is compatible with the picture element will ignore the `img` element, which will be used only by noncompatible browsers.

SVG

Standard Vector Graphics is an open XML specification describing 2D vector graphics. An SVG document can be static (declared in an XML file or tag) or dynamically generated from a JavaScript script using the DOM API. As a vector-rendering engine, a great feature is the adaptation to different screen sizes and resolutions without loss of quality.

SVG is a W3C standard for desktop platforms, with two subsets prepared for mobile platforms: *SVG Basic* and *SVG Tiny*. Thanks to this standards fight (as with XHTML mobile versions), we can use either SVG Basic or SVG Tiny with the same code and results.

The latest version available is SVG Tiny 1.2, but the most compatible version of SVG Tiny for mobile browsers is 1.1 (*SVGT 1.1*), which offers some support for animation. This version has been adopted by the OMA and the 3rd Generation Partnership Project (3GPP). Opacity and gradients are not part of the mobile standard, but some devices still render them. This addition is known in the market as *SVGT 1.1+*.



Modernizr will give you two important flags for compatibility checks, `svg` and `inlinesvg`. WURFL, on the other hand, has the capabilities `svgt_1_1` and `svgt_1_1_plus` to detect SVGT and SVGT 1.1+ support.

One compatibility issue between SVGT devices is text support. Some devices allow us to use system fonts to declare text in the SVG, but others do not, forcing us to convert text to curves in a graphic design tool.



One of the biggest problems of using SVG today is the lack of support in Android browser 2.x, which still has a big portion of the Android market share.

Smartphones and tablets support the last non-Tiny SVG standard without any issue.

SVG can be used in different ways:

- As an external file using `img`
- As an external file using `object`
- As a CSS background
- As a CSS effect
- As a font
- As inline SVG

The *New York Times* uses a GIF logo on its desktop website (<http://nytimes.com>). However, when the desktop website is being browsed by a smartphone or tablet, the logo is replaced with an SVG version (Figure 10-4) so it is sharp at any zoom level and different resolutions, such as on the original iPad and iPad third generation.



Figure 10-4. Because The New York Times uses an SVG logo, there is no loss of quality when the user zooms in or when the site is browsed by a high-density device.

Tools for SVG

Most vector graphic design software supports SVG as an export format. Adobe Illustrator, for example, can export to SVGT 1.1, SVGT 1.1+, and SVGT 1.2. Corel Draw is another useful tool for SVG conversion. Sometimes the markup generated by Illustrator has more tags than you need, so you may want to open it with another tool and export it again. The most useful tool for mobile SVG currently on the market is Ikivo Animator. It is intended for mobile devices and can create animations using SVGT 1.1. You can download a trial version from Ikivo (<http://www.ikivo.com/animator>).



Google Maps uses SVG on the Street View HTML5 version. When you view a street view panorama on Google Maps (<http://maps.google.com>) on a mobile device, the panorama is a bitmap and the direction arrows and street names are a transparent SVG image laid over the top. It's a vector image, so it's easy to rotate and change properties as you browse in the panorama.

In the open source world, Inkscape (<http://inkscape.org>) offers SVG support, and for Windows only we can use SVGmaker Tiny (<http://svgmaker.com>), a printer driver that converts any printed document into SVGT.

In the online world, we can find free SVG editors (<http://code.google.com/p/svg-edit>).



Google Swiffy (<https://www.google.com/doubleclick/studio/swiffy/>) is an online tool that converts Flash movies (ActionScript 2 and experimental ActionScript 3) to SVG and JavaScript.

The recommendations for SVG document generation are:

- Keep the quantity and size of the objects used to the minimum.
- Avoid big gradient areas; they decrease performance.
- Reduce path points to the minimum.
- If the object is too complex, a raster PNG may be better.
- Use GZIP if compatible.
- Combine paths when possible.
- Maintain one copy of each object online.
- Export text as curves.
- For simple shapes, don't use a graphic design tool.

SVG for beginners

SVG is beyond the scope of this book, but here is a very quick lesson on SVG Tiny.

An SVG document is an XML document with a root `svg` tag defining an original view-port size (width and height), but remember that it is a vector image, so you can resize it. Inside the image, we can draw the following kinds of shapes:

- Rectangles (`rect`)
- Circles (`circle`)
- Ellipses (`ellipse`)
- Lines (`line`)
- Polylines (`polyline`)
- Polygons (`polygon`)
- Paths (`path`)

The following is an SVG Tiny 1.1+ compatible document (it uses a LinearGradient, which is not included in the SVGT 1.1 standard):

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1 Tiny//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11-tiny.dtd">

<svg version="1.1" baseProfile="tiny"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    x="0px" y="0px" width="200px" height="200px">

  <linearGradient id="grad1" gradientUnits="userSpaceOnUse"
    x1="54" y1="61" x2="147" y2="61">
    <stop offset="0" style="stop-color:#FFFFFF"/>
    <stop offset="1" style="stop-color:#000000"/>
  </linearGradient>

  <rect x="0" y="0" fill="url(#grad1)" stroke="#000000"
    width="193" height="84"/>

  <ellipse fill="#FF0000" stroke="#000000" cx="30" cy="100"
    rx="25" ry="25"/>

</svg>
```

This document creates a 200×200-pixel SVG image containing a rectangle (rect tag) and a circle (ellipse tag with equal radius rx and ry centered at cx, cy). The circle is filled with a plain red color and the rectangle is filled with a linear gradient defined with an id of grad1. This gradient is not compatible with SVGT 1.1 (without the +).



Google Drive (<http://drive.google.com>) has an online free vector graphic designer for diagrams and graphs. It has the option to export to SVG.

This document has a size of 670 bytes (0.6 KB) as an SVG file. A 24-bit PNG with the same image has a size of 5.77 KB, and an 8-bit PNG with quality loss has a size of 1.31 KB. Clearly, SVG is better from a size (and network traffic) perspective. The bad thing is that the browser has to render the image on the mobile device (although this is not the case for cloud-based browsers, such as Opera Mini, where the SVG is pre-rendered on a server).

Embedding the SVG in XHTML

To insert an SVG document inside an XHTML document, we can use the object tag, defining the data attribute with the URL of the SVG, the type as image/svg+xml, and the width and height. As an SVG is a vector-based image, we can use percentages for

the size attributes to adapt the content to the viewport size. If SVG is not available, we can use the fallback feature to create an alternative image in another format as a child of the object tag:

```
<object data="logo.svg" type="image/svg+xml" width="100" height="30">
  
</object>
```

Embedding the SVG using img

Some HTML5 browsers (as seen in Table 10-1) also use the classic `img` tag for SVG files:

```

```



Filter Effects (<http://mobilexweb.com/go/w3cfilter>) is a W3C draft that allows us to use simple effects or SVG effects on DOM elements. At the time of this writing there is no mobile browser supporting SVG effects, but it's possible that we will see this kind of compatibility in the future.

Inline SVG

HTML5 offers the ability to embed an SVG document using an `svg` element. Inside the `svg` element we must follow XML rules. With this approach we don't need to load an external file, as we do when using `object` or `img`. The following HTML5 document is valid and includes the previous SVG drawing:

```
<!DOCTYPE html>
<title>SVG demo</title>

<h1>SVG demo</h1>

<svg xmlns="http://www.w3.org/2000/svg" id="svg1" version="1.1"
  viewBox="0 0 100 100"
  preserveAspectRatio="xMidYMid slice" width="300" height="300">

  <linearGradient id="grad1" gradientUnits="userSpaceOnUse"
    x1="54" y1="61" x2="147" y2="61">
    <stop offset="0" style="stop-color:#FFFFFF"/>
    <stop offset="1" style="stop-color:#000000"/>
  </linearGradient>

  <rect x="0" y="0" fill="url(#grad1)" stroke="#000000"
    width="193" height="84"/>

  <ellipse fill="#FF0000" stroke="#000000" cx="30" cy="100"
    rx="25" ry="25"/>
</svg>
```

Figure 10-5 shows the result.



Figure 10-5. With inline SVG we can embed a vector graphic inside any HTML5 document on compatible browsers.

With SVG we can use the DOM API to manage elements, such as creating new SVG elements, or changing elements or attributes. For example, this JavaScript code will add a circle over the svg element with the ID `svg1`:

```
var newCircle = document.createElementNS('http://www.w3.org/2000/svg',
    'circle');
newCircle.setAttribute('r',20);
newCircle.setAttribute('id','leftEye');
newCircle.setAttribute('cx',50);
newCircle.setAttribute('cy',40);
newCircle.setAttribute('fill','red');
newCircle.setAttribute('stroke','white');
document.getElementById('svg1').appendChild(newCircle);
```

Raphaël

While SVG is getting more important in the HTML5 world for vector-based graphics, lots of other frameworks are available. One of the most important is *Raphaël* (<http://dmitrybaranovskiy.github.io/raphael>).

Raphaël can create really cool graphics, animations, and effects with just a few lines of code. Once you add the Raphael.js library to your HTML project, you can start using the API, as in this example:

```
// Creates 320 × 200 canvas at 10, 50
var paper = Raphael(10, 50, 320, 200);

// Creates circle at x = 50, y = 40, with radius 10
var circle = paper.circle(50, 40, 10);

// Sets the fill attribute of the circle to red (#f00)
circle.attr("fill", "#f00");

// Sets the stroke attribute of the circle to white
circle.attr("stroke", "#fff");
```



In Chapter 11, we will talk about using custom fonts, and we will see how to use SVG as a font format.

Canvas

The possibility of a drawing API was the dream of every web designer in the '90s, but it has only recently become available in browsers. `canvas` is an HTML5 element that defines a rectangular area where we can draw using a JavaScript API. The drawings are not vector-based, and we cannot browse through them using the DOM or any other mechanism. `canvas` is not a competitor for SVG; it's just another way to generate dynamic graphics in a browser window.



`canvas` was originally defined by Apple in WebKit, and today it is the oldest and the most widely adopted HTML5 feature in the web developers' world. You will find a lot of resources on the Web on the usage of this element.

For our purposes, the only useful attributes are an `id` and the dimensions `width` and `height`. As child elements, we can insert any HTML code that will be used as a fallback:

```
<canvas width="300" height="300" id="canvas">
  Here goes text, images, or other tags for noncompatible browsers
</canvas>
```

The context

Once we have defined a canvas we get what is called a *2D context*: a JavaScript object that we can use for drawing bitmaps over that canvas.

We can get the context pointer using the following code, with the code checking for API support first:

```
var canvas = document.getElementById('canvas');
if (canvas.getContext) {
    // canvas is supported
    var context = canvas.getContext('2d');
}
```

Lines and strokes

Once we have the context, we can define the line type using the color properties `fillStyle` and `strokeStyle` and the integer property `lineWidth`. Then we can start drawing.

Drawing methods

The available drawing methods of the 2D context are listed in Table 10-2.

Table 10-2. Useful drawing methods in the HTML5 canvas context

Method	Description
<code>fillRect(x, y, width, height)</code>	Draws a filled rectangle with the current styles.
<code>strokeRect(x, y, width, height)</code>	Draws a stroked rectangle with a transparent fill.
<code>clearRect(x, y, width, height)</code>	Clears the area and makes it transparent.
<code>beginPath()</code>	Begins a path drawing.
<code>closePath()</code>	Closes the shape by creating a line from the first path line to the ending path line.
<code>moveTo(x, y)</code>	Moves the pen to the coordinates for the next line in the path.
<code>lineTo(x, y)</code>	Draws a line from the current pen coordinates to the ones provided.
<code>arc(x, y, radius, startAngle, endAngle, anticlockwise)</code>	Draws an arc with its center at <i>x, y</i> and with the defined radius. The <i>anticlockwise</i> parameter is a Boolean value. Angles are defined in radians.
<code>quadraticCurveTo(controlx, controly, x, y)</code>	Draws a quadratic Bezier curve.
<code>bezierCurveTo(control1x, control1y, control2x, control2y, x, y)</code>	Draws a cubic Bezier curve.
<code>stroke()</code>	Draws the path defined since the last <code>beginPath()</code> .
<code>fill()</code>	Closes the path defined since the last <code>beginPath()</code> and fills it.
<code>drawImage(x, y)</code>	Draws an image (Image JavaScript object) on the canvas. Other optional parameters also exist.
<code>createImageData(width, height)</code>	Creates an <code>ImageData</code> object with a <code>data</code> attribute that is an array of pixels to be manipulated as integers.
<code>getImageData(x, y, w, h)</code>	Gets an <code>ImageData</code> object from the current drawing to be manipulated.
<code>putImageData(image_data, x, y)</code>	Puts an <code>ImageData</code> object into the drawing.

Method	Description
<code>strokeText(string, x, y)</code>	Draws a stroked string.
<code>fillText(string, x, y)</code>	Fills a string.

canvas as a whole is outside the scope of this book. If you want to dig deeper into this API, you can read the book *HTML5 Canvas* by Steve Fulton and Jeff Fulton (O'Reilly).

An excellent example of drawing curves on a canvas is provided by Mozilla's documentation, which shows how to draw a dialog box with the following code:

```
context.beginPath();
context.moveTo(75,25);
context.quadraticCurveTo(25,25,25,62.5);
context.quadraticCurveTo(25,100,50,100);
context.quadraticCurveTo(50,120,30,125);
context.quadraticCurveTo(60,120,65,100);
context.quadraticCurveTo(125,100,125,62.5);
context.quadraticCurveTo(125,25,75,25);
context.stroke();
```



Angles in the Canvas API are defined in radians, not degrees. To make the conversion we can use the formula $(\text{Math.PI}/180) * \text{degrees}$.

Advanced features

Some other advanced features for canvases include a Text API (to draw text), text shadowing, gradients, image scaling, transparency, fonts, line styles, patterns, and other drawing methods.



In WURFL, the property that we can query server-side is `canvas_support`; in Modernizr, we have `canvas` and `canvastext` for the Canvas Text API.

With canvas we can do lots of cool stuff in mobile web apps, including creating games (using a game loop), drawing apps, charts, and more. There are hundreds of frameworks and libraries out there that we can use as a high-level API instead of talking directly to the Canvas API.

Canvas as native code

Ejecta (<https://github.com/phoboslab/Ejecta>) is an open source iOS Canvas implementation allowing us to create high-performance native Canvas apps and games without a web view and without the HTML rendering engine being in memory. It's possible that

in the next few years we will see this kind of solution available across platforms, allowing the creation of high-performance HTML5 games that will rely not on an HTML/CSS rendering engine but only on a JavaScript execution engine and a native Canvas implementation on each platform.



At the time of this writing, Canvas 2.0 (<http://www.w3.org/TR/2dcontext2/>)—also known as HTML Canvas 2D Context, Level 2—is being discussed as a draft specification. As yet, there is no browser implementation.

Canvas as a background

WebKit has released an extension to the Canvas API that allows us to define a drawing context as a CSS background instead of a canvas element. To do that we need to first define the background in CSS as `-webkit-canvas(id)` and use a `document.getCSSCanvasContext` function to retrieve the context. That function will receive the context type (2d), the `id` defined in CSS, and the desired width and height.

For example:

```
<!DOCTYPE html>
<title>Canvas Background</title>
<style>
#element {
    width: 200px; height: 100px;
    background: -webkit-canvas(myCanvas);
}
</style>

<div id="element"></div>

<script>
var context = document.getCSSCanvasContext("2d", "myCanvas", 200, 100);
// All my drawings here
</script>
```

At the time of this writing, this ability is only available on modern WebKit-based mobile browsers, such as the Android browser, Safari on iOS, BlackBerry 7/10, and Chrome for Android.



On Firefox, we can achieve a similar effect using `background: -moz-element(#canvasId)` and a hidden canvas element on the screen connected though the `id`.

Exporting canvas drawings

Another extension of the canvas element is the ability to export the canvas drawing as a bitmap image in data URI format. With this API, we can then use Ajax to send the drawing to the server, or use a storage API to store it locally. Remember that the basic Canvas API already has a way to export the drawing as an array of pixels. This API will export the bitmap as a real, valid PNG file, unless we define a different MIME type as a parameter. When we define JPEG as the result format, we can use a second argument (from 0.0 to 1.0) indicating the JPEG compression value.

The method used is `toDataURL`, and it can be applied to the canvas element itself:

```
var bitmapDataPNG = canvas.toDataURL();  
var bitmapDataGIF = canvas.toDataURL("image/gif");  
var bitmapDataJPG = canvas.toDataURL("image/jpeg", 0.8);
```



Exporting the canvas is particularly useful when the drawing is created by the user, or when we want to provide a local way to manipulate an image and export a new version, such as applying a bitmap effect, resizing an original image, or some other manipulations.

Some modern browsers also support a way to get the image as a Blob object using `toBlob()` or as a File API object using `getAsFile()` (this function is prefixed, as in `mozGetAsFile()`). You should check the context object for the presence of these functions before trying to use them.



Adobe Flash Professional (<http://adobe.ly/Z6aJHX>) from version CS6 supports the CreateJS toolkit extension that helps Flash animators to export assets, animations, and drawings to the Canvas 2D drawing API using the open source CreateJS framework.

Canvas and pixel density

As we already know, we need to deal with screens having different pixel densities, and when we create a 300×200-pixel canvas, it's measured in CSS pixels. On most mobile platforms, canvases are considered as bitmap images, so by default we will get a resized image that will not use the full definition of the device. For example, Safari on iPhone 5 will render your drawing as a 300×200 bitmap and then resize it to 600×400, with the consequent quality loss. Safari on the Mac with a Retina display acts differently, though: the canvas will use all the possible definition available.

To manage these differences, WebKit has added a new property to the canvas context, called `webkitBackingStorePixelRatio`. This means that you will get a device pixel ratio

of 2 on both a Mac with a Retina display and an iPhone 5, but on iOS you will get a backing store pixel ratio of 1.

If we want to force a high-definition canvas, one solution is to double the canvas size in HTML and also the viewport width (for example, to 640 pixels on the iPhone or iPod).



Pay attention to the performance, as when you double the canvas size, your rendering will take more processing time.

Another solution to force a high-definition canvas is to force a double-size canvas. Canvases have a virtual dimension that usually is 1:1 with CSS pixels. We can change that using the `scale()` function of the context. That is, if we say:

```
context.scale(2, 2);
```

our canvas will be double the size. The problem is that the drawing will also be double the size in our container. To solve this problem, we need a trick. The solution is to define the canvas dimensions in HTML (as double the desired size) and also in CSS (as the desired size):

```
<canvas width="600" height="600" style="width: 300px; height: 300px">
```

With that solution we will finally have a 300×300 CSS pixels canvas that is being rendered as a 600×600 real pixels image on high-resolution devices.



Be careful to apply the high-resolution techniques only for devices with a high-density screen and when you know that the canvas is not being doubled in size automatically (as with the backing store pixel ratio value).

WebGL

WebGL (<http://www.khronos.org/registry/webgl/specs/1.0>) brings 3D to the mobile web through a canvas element. The API is not a W3C standard; it was created by the Khronos Group. The API is based on OpenGL ES 2.0, and to be honest, it's not too webby. That means that the API doesn't look like a JavaScript API, but more like a C++ API with JavaScript syntax.

You need development knowledge to work with WebGL because it's not as simple as working with the 2D standard Canvas API. We use the same canvas element and we just change the context we look for:

```
var context = canvas.getContext("webgl") ||  
               canvas.getContext("experimental-webgl");
```

Some browsers will use `webgl` and others `experimental-webgl` as the context name. With the context we can access the graphics or video card directly from JavaScript.



Even when a mobile browser supports WebGL, it doesn't mean that the performance will be good enough for some kinds of 3D content. Because of performance and some security issues, some browsers ask the user for permission before starting WebGL sessions, or allow the user to disable it from the preferences.

WebGL compatibility on mobile browsers is quite rare today, being limited to the BlackBerry browser for PlayBook and BB10, Firefox, and Opera for Android from version 12. At the time of this writing there is no support in Chrome, the Android browser, Internet Explorer, or Safari on iOS. The only exception is some Sony Xperia devices on Android 2.3 that have added WebGL support on top of the default browser.



Safari on iOS from version 4.2 includes all the necessary properties and objects of WebGL in the engine, but it's completely disabled for developers. iAds, the advertisement system from Apple for native iOS apps, can make use of WebGL for 3D content under Apple's approval.

If you have a compatible browser, you can check out the open source TunnelTilt (<http://blackberry.github.com/WebGL-Samples/tunneltilt>) game created by BlackBerry. The source code is available at BlackBerry's GitHub (<https://github.com/blackberry/WebGL-Samples>).

Because the WebGL API accesses the video card as a low-level API, some security issues have been found that allow a website to potentially take information from other windows that the user may have open. That is why the BlackBerry browser for PlayBook shows a security warning before enabling WebGL, as seen in Figure 10-6.

For more information on WebGL, check out the book *WebGL: Up and Running* by Tony Parisi (O'Reilly).

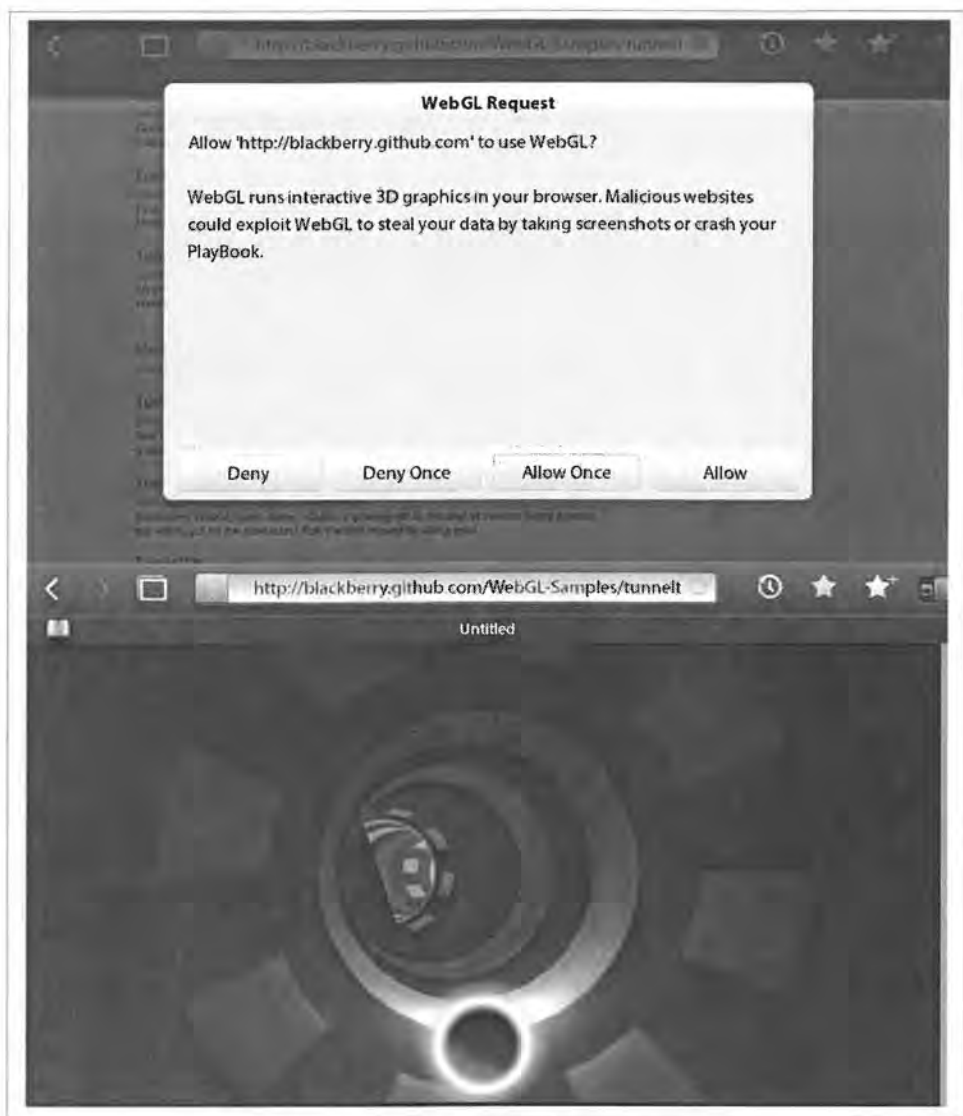


Figure 10-6. The BlackBerry browser for PlayBook shows a security warning before starting a WebGL 3D session (in this case, the TiltTunnel game).

Canvas compatibility

Table 10-3 lists browser support for basic canvas functionality. For updated compatibility information, please visit Mobile HTML5 (<http://mobilehtml5.org>).

Table 10-3. HTML5 canvas compatibility table

Browser/platform	2D canvas	High-resolution canvas	Canvas as background	Export to data URI	WebGL
Safari on iOS	Yes	No	Yes	Yes	No
Android browser	Yes	No	Yes	Yes from 4.0	No
Chrome for Android	Yes	No	Yes	Yes	No
Nokia Browser for Symbian	Yes from Anna	N/A	No	No	No
Nokia Xpress browser	No (only exported as data URI)	No	No	Yes	No
BlackBerry browser for smartphones until 7.x	Yes from 6.0	N/A	Yes	Yes	No
BlackBerry for PlayBook	Yes	N/A	Yes	Yes	Yes from 2.0
BlackBerry for BB10	Yes	No	Yes	Yes	Yes
Internet Explorer	Yes from 9	No	No	No	No
Firefox Mobile	Yes	No	No	Yes	Yes
Opera Mobile	Yes	No	No	Yes	Yes from 12.0
Opera Mini	Yes, rendered on the server	No	No	Yes	No
Amazon Silk	Yes	No	Yes	No	No
UC browser	Yes	No	Yes	No	No

Adobe Flash

The Flash Player has long been the de facto standard in desktop browsers: Flash Player 8 penetration was at more than 99.5% at its peak in September 2009. However, the world has changed, and mobile devices are part of the change. While Adobe has tried to bring the same experience to the mobile browser, it has found many stones in the road.

Adobe had two mobile lines: Flash Lite and the major Flash Player for mobile devices. The first one was intended for low-end and mid-range devices (and high-end devices shipped before 2010), and the major player was available for Android and BlackBerry for PlayBook starting in 2010. However, after a couple of months it became clear that the experience and performance on Android were not good enough, and finally Adobe gave up and started to support HTML5 as the way to go.



Flash Player for Android was available from Android 2.2 to Android 4.0. With Jelly Bean (4.1), Adobe has removed support, and it will not invest in enhancements to the already deployed versions of the player.

At the time of this writing only the BlackBerry PlayBook tablet and the BlackBerry 10 platform have the full Adobe Flash Player; all modern Symbian devices support Flash Lite 4 (similar to Flash Player 8 in terms of compatibility). Internet Explorer 10 on Windows 8 supports Flash in Desktop mode (not available on ARM tablets); in Windows 8 mode it is disabled by default.



The future of Flash on mobile devices is about creating native apps for iOS, Android, and BlackBerry 10 using Adobe AIR, not about mobile websites. Use HTML5 for the browser and multiplatform native web apps.

If your website really needs to use Flash content for Internet Explorer 10, you can add a header:

```
X-UA-Compatible: requiresActiveX=true
```

or a meta tag:

```
<meta http-equiv="X-UA-Compatible" content="requiresActiveX=true" />
```

indicating that it is required. Either of these will invite the Internet Explorer to open the website in Desktop mode in Windows 8, so your Flash content will run properly.



Mozilla Shumway (<http://mozilla.github.com/shumway/>) is an open source Flash Player completely written in JavaScript. It can run SWF files with ActionScript 3 on HTML5 browsers. The project is based on an older one known as Gordon.

Also, as long as your website does not make use of certain APIs (such as 3D or hardware access), you can request that Microsoft add you to a whitelist so Flash Player will be activated in IE10 Windows 8 mode on your website. More information is available at Breaking the Mobile Web (<http://mobilexweb.com/go/win8flash>).

Apple Versus Adobe

Apple versus Adobe, Adobe versus Apple...this is one of the great fights in the mobile web world today. When the iPhone SDK arrived in 2008, Adobe wanted to create a Flash Player for it. Apple appeared to agree. A year later, Adobe accused Apple of not providing any help, and months later Apple announced that the iPhone would not have Flash support. The justification was the high battery consumption of Flash content in the browser, and the fact that it was not necessary because iPhone extensions to CSS and JavaScript allow any developer to create Flashy content without Flash.

One of the off-the-record causes, though, was that if Apple enabled Flash content in the iPhone, it would lose political control over the content, games, and applications available for the device.

In early 2010, Adobe announced that starting with Adobe Flash CS5, the tool would export SWF to native iPhone applications that could then be distributed in the App Store. However, Apple counterattacked by changing the terms of the App Store and saying that those applications would not be allowed. A year after that, Apple removed the “anti-Flash” restriction and Flash applications started to be accepted again at the App Store.

The issue of Flash compatibility is an important one because of the undesirable results when we attempt to show Flash content on noncompatible devices (as illustrated in Figure 10-7).



Figure 10-7. As we can see, showing Flash content on noncompatible devices is a big mistake.



Microsoft Silverlight had a short life in the desktop world as a browser plug-in. In the mobile world it was only a flash in the pan, with beta versions for Windows Mobile and Symbian released in 2010 that never even graduated to final releases.

Video

Serving video content to mobile devices is very important for many portals and content providers.

We can provide multimedia content in three formats:

- Downloadable content
- On-demand streaming content
- Live streaming content

Before we look at any code, let's talk about the codec hell.

Containers and Codecs

Video and audio files come with two technologies: a container format and one or more codecs inside. The container is what you usually see more frequently: the file extension, such as *.mov* or *.mpeg*. The codec is the algorithm that encodes and decodes the media; these are usually divided into video and audio codecs.

MPEG-4 containers are the most common. For these containers, AAC is the typical audio option, and the most compatible video codecs are H.263 and H.264; of the two, H.264 is preferred because of its compression algorithm (you get a smaller file for the same quality). The combination of MPEG-4/H.264/AAC is the most compatible solution for today's smartphones, social devices, and tablets. However, even when we say "H.264" there are several different configurations that the video can use, and not every one is compatible with every device, so testing is really important.



It's important to understand the difference between containers and codecs. When you talk about "an MP4 video," you are not saying anything about the audio and video codecs involved, which may be incompatible with some browsers.

There are also devices with support for other containers, such as Flash Video (FLV), Audio Video Interleave (AVI), Real Video (RV), QuickTime Movie (MOV), and Windows Media Video (WMV) containers, but compatibility varies and using them is not a good idea in terms of multiplatform accessibility.



H.264, also known as AVC, is one of the most used codecs and is commonly distributed inside MPEG-4 files. Its power is defined by its ability to provide high quality with smaller sizes. The problem is that its use also involves some royalties, which caused some desktop browsers to remove support for this codec. This isn't an issue on mobile devices because the video playing is usually managed by the operating system, so H.264 is well supported on most platforms.

Because of the proprietary nature of MPEG-4 and its associated codecs, the community has been working on some open and free alternatives over the last few years. The mobile community also created a container specifically designed for mobile devices a few years ago: the 3GP container.

The MPEG-4/H.264 alternatives include:

- The *Ogg* container, which usually comes with a Theora free video codec and a Vorbis audio codec (sometimes known as Ogg/Theora/Vorbis). On desktop, it's compatible with Firefox, Opera, and Google Chrome but on mobile it's right now only on Firefox.
- The *WebM* container, open and free project created by Google for HTML5 era. This container uses a VP8 video codec and Vorbis for audio. On desktop, Firefox, Opera, and Google Chrome support it, and on mobile only Android Browser from 2.3 and Google Chrome for Android.
- The *3GP* and *3GP2* containers, created by the 3GPP organization. These are based on MPEG-4 and optimized for mobile devices; they are still well distributed on mobile devices, but MPEG-4 is a much more modern solution and is referred today.



Handbrake (<http://handbrake.fr>) is a free and open source video transcoder for Mac OS, Linux, and Windows that will help us with video conversions for mobile devices.

Reference movies for iOS

Safari on iOS also supports "reference movies," created with QuickTime Pro or a similar tool. A reference movie provides a list of movie URLs with different bit rates (for example, for WiFi, 3G, or EDGE), so QuickTime can select the correct one for the device.



You can find an Objective-C Mac tool (<http://www.mobilexweb.com/go/refmovie>) provided by Apple to generate iOS reference movie files from the command line.

Delivering Video

We can deliver files using HTTP or a multimedia-streaming server, such as Adobe Flash Media Server. Unfortunately, most mobile browsers only support HTTP as the transport layer.



Multimedia files are generally large. If we deliver noncompatible files, the user will be paying for useless traffic and will not be happy with us.

Check your server configuration, because using HTTP 1.1 we can support partial downloads and the ability to seek to any part of the file. For the best HTTP streaming technique, we need the server to support partial downloads. If your server doesn't support that, there is a great PHP script (<http://www.mobilexweb.com/go/phppartial>) available.

Linking to video files

Usually, if MIME types are well defined on the server side, the user clicking on a link to a video file will start a video download automatically. On modern devices, instead of downloading the file, the browser will open the default video player application, which (if both client and server are compatible) will start playing your file using HTTP streaming. Only older feature phones will not play the video until the complete file has been downloaded.



WURFL and Modernizr provide capabilities to help you understand video container and codec compatibility on both the server and the client side.

The HTML5 video Element

HTML5 includes a new video element that can be used to embed and/or play videos in full-screen mode. We must specify a width, a height, and one or more video files to play. We can define one source video file using the `src` attribute, as in:

```
<video width="300" height="200" src="video.mp4"></video>
```

Because of the codec hell discussed earlier, the video element accepts a multiple-video definition using child source elements. That is, we can define different videos using different containers and codecs, and the browser will take the first one that is compatible with the current environment:

```
<video width="300" height="200">
  <source src="video.mp4" type="video/mp4">
  <source src="video.ogv" type="video/ogg">
```

```
<source src="video.webm" type="video/webm">
</video>
```

The type attribute can contain just the MIME type for the container (such as video/mp4), or it can also include codec version declarations, such as:

```
<source src='video.mp4' type='video/mp4; codecs="avc1.58A01E,
mp4a.40.2"'>
```

The video element typically acts as a video player in a rectangular area on the HTML page. However, on most smartphones it simply acts as a placeholder on the page; when we start playing the video, it always moves to a full-screen mode managed by the media player app, such as QuickTime on iOS. We can see this in action in Figure 10-8.



Figure 10-8. A video element is shown as a placeholder that will open a full-screen player on some mobile browsers (in this case, Safari on iOS 6, Internet Explorer 9, and Android browser 2.3)—in some cases, the playing is in landscape mode.



Safari on iOS will only show the Play icon over the placeholder after loading the video metadata, so it knows that the video can effectively be played. If the video can't be played or doesn't exist, an error icon is shown instead.

On tablets, the video element acts similar to on a desktop: the video is rendered inside the HTML page and not in fullscreen mode, as on smartphones. Figure 10-9 illustrates.

HTML5 video



HTML5 video



Figure 10-9. On tablets, the video is usually embedded in the HTML page with a manual option to change to full-screen.

By default most browsers will add a black background to the video area when the video is not playing. We can change that using the `poster` attribute, which defines an image to be displayed before the video starts playing:

```
<video width="300" height="200" src="video.mp4" poster="video_poster.png">
</video>
```

You don't need to add a Play button image inside the poster picture, as most browsers will add a custom icon by default (as we can see in Figure 10-10).

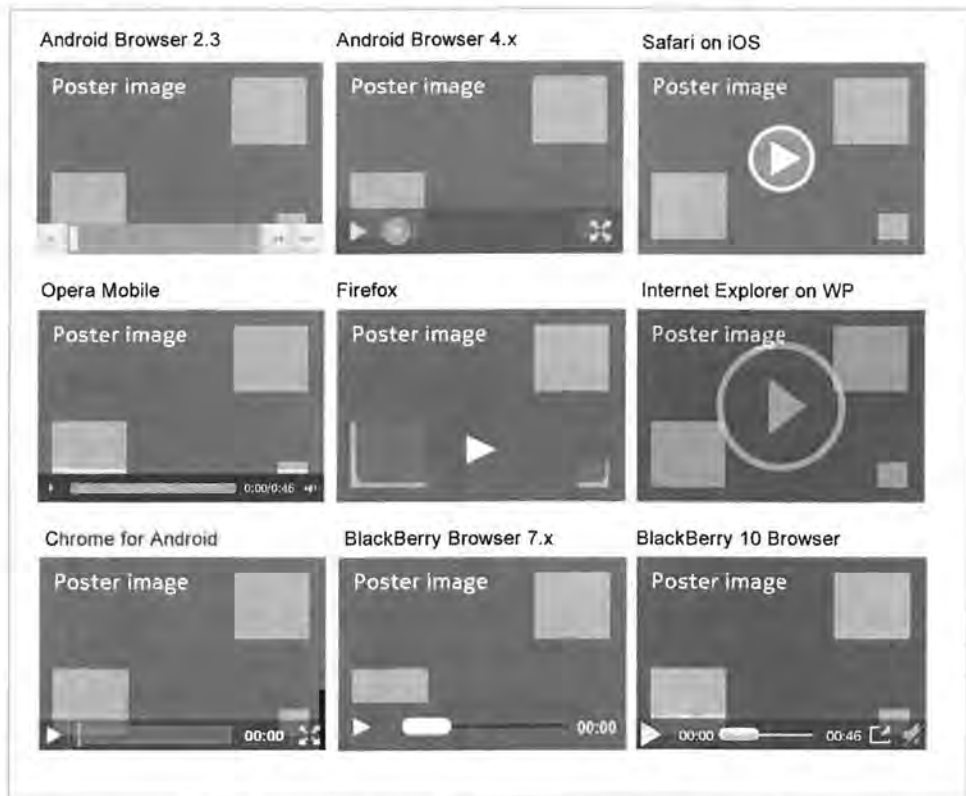


Figure 10-10. When we define a poster image, we should not add a Play icon; most browsers add one automatically.



The HTML5 video element supports one or more track child elements that will define Closed Captions or subtitles for the video. However, at the time of this writing no mobile browser supports this.

For devices that will show the video embedded in the page, we have to use the Boolean controls attribute that, if present, forces the browser to show playing controls (such as Play, Pause, Stop, volume sliders, or track elements). On smartphones we will always have these controls, and there is no way to remove them. On some tablets, we have the option to use default controls or provide our own (see Table 10-4, later in this section).

For example, on the iPad, if we want a video player to have controls we must define the controls attribute:

```
<video width="300" height="200" src="video.mp4" controls></video>
```



On most mobile browsers there is no way to play more than one video at the same time.

Advanced declarations

The video element supports more attributes that we can define. However, support on mobile browsers is quite rare, as the browser or the media player application will take control of the player, and not our HTML code. These attributes are:

autoplay

A Boolean attribute that, if present, will start the video playing automatically. On mobile browsers it's usually disabled as the playing action will be triggered only by a user action.

loop

A Boolean attribute that, if present, will start playing the video from the beginning again when it reaches the end.

muted

A Boolean attribute that, if present, will start the video in silent mode.

preload

Defines how to buffer or preload the video. It accepts the values none, metadata, or auto.



On desktop browsers we can apply any CSS attribute to the video player, such as transformations, shadows, or transitions, and we can also use a video player as a canvas context image. However, because of the nature of video playing on mobile browsers (only full-screen or as a plug-in), these browsers do not support these features.

Providing fallbacks

Anything inside the video element, apart from the source elements, will be ignored by HTML5-compatible browsers but used by noncompatible browsers, so we can, for example, add a message, a link to the video, or an alternative embedded system, such as object, embed, or Flash:

```
<video width="300" height="200" src="video.mp4">
  <!-- Fallback HTML code -->
  Embedding video is not compatible with your browser.
  <a href="video.mp4">Download the video</a>
</video>
```



Flash Lite (from version 3.0) supports only Flash Video format (FLV), while the major Flash Player supports FLV and MPEG-4. We can use the Flash Player as a fallback if HTML5 video is not available.

Video player API

HTML5 browsers include a video player API that we can use to manage video playing from JavaScript. This API usually involves events that we can bind to so we can detect video download errors, codec errors, and video playing time events (such as `onplay` and `onpause`). With this API we can also use JavaScript to provide our own video playing controls, as any video DOM element has `play()` and `pause()` methods and properties such as `volume` and `currentTime`. Safari on iOS will ignore any volume definition, as the user can change it using the hardware keys:

```
var player = document.getElementById("player");
player.volume = 0.5;
player.play();
```

The problem with mobile browsers is again the full-screen playing experience. With that problem, our website is frozen while the video is playing, so we have no option to customize the experience unless the user's device is a tablet.

There are a few open source video player solutions available on the market, such as HTML5 VideoJS (<http://videojs.com>) and jPlayer (<http://jplayer.org>).



On most mobile browsers, such as Safari on iOS, we can only execute the `play()` method after a user action, such as a `click` event. Other solutions, such as using `load` events of timers, will be ignored.

Streaming to Apple TV

If the user has an iOS device (iPod, iPad, or iPhone) and also an Apple TV, the set-top box from Apple, we can allow the user to stream a video from her device to the TV automatically. The wireless streaming is called *AirPlay*. To define how our video element should work with AirPlay, Apple offers a nonstandard `x-webkit-airplay` attribute supporting the values `allow` or `deny`.



If we have video playing rights for iPads and iPhones but not for TVs, we can deny AirPlay explicitly. The user will not be happy, but crazy license terms are around us all the time!

To allow streaming to Apple TVs, we should use `x-webkit-airplay="allow"`:

```
<video src="video.mp4" width="300" height="200" controls
      x-webkit-airplay="allow">
</video>
```

When an Apple TV is available on the same network as the iOS device, the user will see the AirPlay TV icon, as shown in Figure 10-11. From iOS 5, AirPlay is enabled by default; before that version AirPlay is disabled by default.



Figure 10-11. When using `x-webkit-airplay="allow"`, we can allow the user to stream the video file to an Apple TV on the same network (the last icon on the right).

Streaming

Streaming audio or video is a difficult solution if we want to be compatible with all devices, because different platforms support different streaming technologies.

Some devices, including Symbian, Windows, and BlackBerry devices, support the Real Time Streaming Protocol (RTSP). When a link with this protocol is used (such as in `rtsp://server/content`), the default media player—Real Player or Windows Media—is opened. The content can be a file to be streamed (a prerecorded audio or video file) or a live event (such as radio or TV show, sports event).

Modern smartphones, such as Android and iOS devices, don't support these streaming standards, which were created for the desktop world and direct connections. Consequently, different streaming solutions have appeared on the market over the last year to use HTTP as a streaming transport protocol. These standards are:

- Apple HTTP Live Streaming (HLS)
- Adobe HTTP Dynamic Streaming
- Microsoft Smooth Streaming
- Dynamic Adaptive Streaming over HTTP (DASH)

Today, the Apple solution seems to be the most compatible with at least some mobile browsers.

HTTP Live Streaming

Apple has created a new way to deliver live streaming using HTTP, called HTTP Live Streaming, which it has presented to the IETF as a proposed Internet standard. It is supported from iOS 3.0, and on Android since 3.0, and allows the transmission of live video (such as a live event) using the same HTTP method we know. In fact, this is the only streaming solution that works on iOS, webOS, and Android devices using the video element.

Implementing HTTP Live Streaming requires some changes on the web server end. The simplified explanation of the protocol is that on the server, the live stream is buffered in little packages that are sent to the client. It's like transmitting a live radio show by sending a series of 10-second MP3s.



The well-known Akamai application acceleration service provider offers live streaming services for mobile devices from the Akamai iPhone Showcase (<http://iphone.akamai.com>). Influxis (<http://www.influxis.com>) also offers mobile streaming services as a shared hosting solution, for the iPhone/iPad and BlackBerry and Android devices. Cloud-based solutions such as Vimeo also support video streaming on mobile devices.

HTTP Live Streaming supports the H.264 codec for video and AAC or MP3 for live audio streaming, as well as a bandwidth switcher for different qualities. However, the best feature is that it passes any firewall or proxy because it is HTTP-based.

The implementation involves an extended M3U file (*.m3u8*) including links to different streaming segments, supporting different bandwidths. For example:

```
#EXTM3U
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=688301
http://qthttp.apple.com.edgesuite.net/1010qwoeiuryfg/0640_vod.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=165135
http://qthttp.apple.com.edgesuite.net/1010qwoeiuryfg/0150_vod.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1, BANDWIDTH=262346
http://qthttp.apple.com.edgesuite.net/1010qwoeiuryfg/0240_vod.m3u8
```

The video element will point to that file:

```
<video width="300" height="200" controls src="live.m3u8"></video>
```

Apple offers a prerelease toolkit for this solution called HTTP Live Streaming Tools, supported by the latest versions of the Flash Media Server, Microsoft IIS Media Services, and many others. You need to be part of the Apple Developer Connection to download it. The open source IceCast Server (<http://www.icecast.org>) also supports iPhone streaming, from version 2.3.2.



For Android, iPhone, and Flash Lite 3 or Flash Player 10 devices, we can safely upload video content to YouTube and embed it in our websites or link to it. The devices will render this content properly using the HTML5 iframe video player or a native YouTube application, if available.

Embedding with object

Before HTML5 appeared on the market, the way to embed a video without any Flash support was to use the `object` tag:

```
<object data="video.mp4" type="video/mp4" width="300" height="300" />
```

The previous example worked on Symbian devices for many years.



The HTML fallback mechanism allows us to use the `video` element as the first option, with a fallback to a Flash Player, a fallback to the `object` element, and finally a fallback to a link to the video file.

An alternative solution was the `embed` tag, preferred for iOS 1 and 2.x (but completely replaced by the `video` tag on newer iOS devices):

```
<embed src="poster.jpg" href="video.m4v" type="video/x-m4v" />
```

Video Compatibility

Table 10-4 shows video compatibility for the main mobile platforms.

Table 10-4. HTML5 video compatibility table

Browser/platform	Preferred delivery	Playing UI	JS API	HLS
Safari on iPhone/iPod	video element	Full-screen	Only for play	Yes
Safari on iPad	video element	Embedded	Yes	Yes
Android browser	video element from 2.2	Full-screen up to 2.3 Embedded from 3.0	Yes	Yes from 3.0
Chrome for Android	video element	Embedded	Yes	Yes
Nokia Browser for Symbian	video element from Anna object element before Anna	Full-screen	No	No
Nokia Xpress browser	File link (no video element)	N/A	No	No
BlackBerry browser	video element from 6.0 File link on 5.x	Embedded	Yes	No
Internet Explorer	video element from 9.0	Full-screen on Windows Phone Embedded on tablets	Yes	No

Browser/platform	Preferred delivery	Playing UI	JS API	HLS
Firefox Mobile	video element	Embedded	Yes	No
Opera Mobile	video element	Embedded	Yes	N/A
Opera Mini	File link (no video element)	N/A	No	No
Amazon Silk	video element	Full-screen	Only for play	No
UC browser	video element	Full-screen	Only for play	No

Audio

Audio is another media element that we can use in our mobile websites. HTML5 includes a new audio element that is almost the same as the video element in terms of attributes and compatibility.

While some mobile browsers support WAV and Vorbis audio files, they all work well with the MP3 audio format. Therefore, we can usually use only one source definition with the MP3 audio file, such as:

```
<audio width="300" height="50" controls src="audio.mp3">
  Audio playing not available.
  <a href="audio.mp3">Download the file</a>
</audio>
```



Before iOS 4, all audio elements played as a full-screen blank page following the video player action. From version 4, audio playing is embedded in the HTML page.

If you want the user to have audio controls on the page, as in Figure 10-12, you should always define the controls attribute. If not, you may consider creating an invisible audio element, as described in the next section.



Figure 10-12. The Safari on iOS and Chrome UIs for the audio element—when using audio we need to use the controls attribute if we want something to render on the screen.

Invisible Audio Player

When it comes to audio, we don't need any visual element on the HTML page in most situations—we just want to play a song or a sound effect for a game. To avoid the onscreen controls, we can create a full audio player from JavaScript without using HTML code. The JavaScript API is the same as for the video element. To load an audio file and play it, we can use the following code:

```
var player = document.createElement("audio");
player.src = "audio.mp3";
player.load(); // You can call play() directly or load it first
player.addEventListener("load", function() {
    player.play();
}, true);
```



SoundJS (<http://www.createjs.com/#!/SoundJS>)—part of the CreateJS family—is a cross-platform audio JavaScript API.

The big problem for games is that usually in mobile browsers you can only play one audio file at a time (as we'll see later, in Table 10-5). To solve that problem, the Web Audio API appeared to enable low-level audio control.

Web Audio API

The Web Audio API made its debut in the mobile web space with Safari on iOS 6. It's a pretty new API created by Google and at the time of this writing is in early draft with the W3C.

As a low-level API, we can use it to play binary audio files, create our own sounds dynamically, or apply audio effects in our web apps. To use this API completely, you need some advanced technical knowledge of audio management.

The API is available through an `AudioContext` global constructor or a prefixed version, as shown here:

```
var context = false;
if (window.AudioContext || window.webkitAudioContext) {
    context = window.AudioContext ? new AudioContext() :
        new webkitAudioContext();
} else {
    // API not available;
}
```

For simple audio playing purposes, the following code will bring an audio file by binary Ajax request and play it:

```

var request = new XMLHttpRequest();
var audioBuffer;
request.open("GET", "myaudio.mp3", true);
request.responseType = "arraybuffer"; // Defines binary file
request.onload = function() {
    context.decodeAudioData(request.response, function(buffer) {
        var audioSource = context.createBufferSource();
        audioSource.buffer = buffer;
        // We connect the audio with speakers
        audioSource.connect(context.destination);
        audioSource.noteOn(0); // Play without delay
    });
};
request.send();

```

We can also create much more complex effects with the Web Audio API, such as mixing two audio files, cross-fading them, applying audio filters, and much more.

Audio Compatibility

Table 10-5 shows audio compatibility for the main mobile web platforms.

Table 10-5. HTML5 audio compatibility table

Browser/platform	Preferred delivery	Playing UI	Invisible audio support	Web Audio API
Safari on iOS	audio element	Embedded from 4.0 Full-screen before 4.0	Yes	Yes from 6.0
Android browser	audio element	Embedded	Yes	No
Chrome for Android	audio element	Embedded	Yes	No (until 18)
Nokia Browser for Symbian	audio element from Anna object before Anna	Embedded	No	No
Nokia Xpress browser	File link (no audio element)	N/A	No	No
BlackBerry browser	audio element	Embedded	Yes	No
Internet Explorer	audio element	Embedded	Yes	No
Firefox Mobile	audio element	Embedded	No	No
Opera Mobile	audio element	Embedded	Yes	No
Opera Mini	File link (no audio element)	N/A	No	No
Amazon Silk	audio element	Embedded	No	No
UC browser	audio element	Embedded	No	No

CSS for Mobile Browsers

In earlier chapters we discussed the many standards in the mobile CSS world and noted the CSS extensions available in WAP CSS. Whether we decide to use CSS 2.1, CSS3, CSS Mobile Profile, WAP CSS, or nonstandard extensions, it will be just the same; we'll use CSS at-rules, CSS selectors, and attributes for those selectors. The standards only tell us which ones are supported. What's more, we will find some browsers that do not render standard styles but do render nonstandard ones.

This chapter is not intended as an introduction to basic CSS; instead, we will focus on features that are specific to or important on mobile devices.

Where to Insert the CSS

The first question to answer is: where should we tell the browser what styles to apply? We have many options:

- `<style>` tags inside the XHTML or HTML markup
- External stylesheets (*.css* files)
- `style` attributes inside the tags

The third option might seem like the most efficient approach, but it is not the best one. That said, there are times when it is useful. For the CSS WAP extensions for form controls described in Chapter 6, for instance, it is easiest to insert inline styles to avoid defining IDs and ID selectors for each control:

```
<input type="text" name="name" style="-wap-input-format: A*a">
```



On BlackBerry devices running Device Software 4.5 or earlier, stylesheets can be disabled from the browser or from a corporate policy.

If the website you are creating is a one-page document, it will be faster to include the CSS in the `<style>` HTML tag to avoid a request and a rendering delay. The other ideal situation for this technique is if your home page is very different from the other pages in your site. Otherwise, odds are good that external stylesheets will help you manage your site more efficiently.

Media Queries

Is one CSS stylesheet adequate for all devices? Maybe. The first factor to consider is whether we are working on a desktop HTML site or a mobile-specific one.

If we decide to use only one HTML site for both desktop and mobile devices, our only option for changing the design and layout is the CSS file, unless we don't want any mobile design adaptation. This situation is a good fit for the `media` attribute.

The CSS standard allows us to define more than one stylesheet for the same document, or one stylesheet with different definitions on different types of media. The `media` attribute was part of the CSS 2.1 standard, and the most-used values for `media` attributes are `screen` (for desktops), `print` (to be applied when the user prints the document), and `handheld` (for... yes, mobile devices). There are also other values, like `tv` and `braille`, but no browsers currently support these.

Great! We've found the solution. We can just define two stylesheets, one for `screen` and one for `handheld`, and all our problems will be solved. The two stylesheets can define different properties for the same elements, and we can even use `display: none` to prevent some elements from being shown on mobile devices:

```
<link rel="stylesheet" type="text/css" media="screen"
      href="desktop.css" />
<link rel="stylesheet" type="text/css" media="handheld"
      href="mobile.css" />
```

However, this “ideal” situation becomes hell when we test it. Many modern mobile browsers rely on `screen` stylesheets because they can render any desktop website (that is, they feel they have power enough to render any website, not just mobile-specific ones). And other browsers use `screen` when they think a site is a desktop website and use `handheld` when they think it is a mobile website, depending on the `DOCTYPE`, a meta tag, or the user's view preferences.



We can't rely on the `media="handheld"` attribute for mobile-specific style definitions.

CSS3 Media Queries

CSS3 comes to our help with *media queries*. These complex media definitions include conditions about different properties, such as screen size or current orientation.

For example, we can say: “Apply this stylesheet for devices with a maximum screen width of 480 pixels.” This will apply to an iPhone, because in landscape mode it has a screen width of 480px and it doesn’t support print, handheld, or any other media type. Here’s how to write this as a conditional media query:

```
<link rel="stylesheet" media="screen and (max-device-width: 480px)"
      href="mobile.css">
```

We can also define CSS media queries inside the same stylesheet file. For example, the following code will change the background color displayed on a mobile device:

```
@media screen and (max-device-width: 480px) {
  body {
    background-color: red;
  }
}
```



If you want to test media queries dynamically in your browser, check out the Media Queries Test (<http://mediaqueriestest.com>) or browse to m.ad.ag on your mobile device.

The standard is defined as a media type and at least one expression. The types are the same ones available in CSS2, including screen, print, handheld, and all. However, most modern browsers accept expressions without any type definition, in which case they apply all and—for example, (width: 480px) is equal to all and (width: 480px).



Defining a media type and using parentheses for expressions are required by the standard. However, most browsers will evaluate the expression without them.

Media queries accept expressions and some operators. Parentheses are required around expressions. The possible types of conditional expressions are:

Boolean attributes

Such as color or monochrome.

Attribute value equal to

The syntax is `<attribute>: <value>` and it is always evaluated as “equal to,” such as `width: 500px` or `density: 326dpi`. The values can accept CSS units and constants.

Attribute greater than or equal to

The syntax is `min-<attribute>: <value>`, such as `min-width: 500px` (meaning “if the width is equal to or greater than 500 CSS pixels”).

Attribute less than or equal to

The syntax is `max-<attribute>: <value>`, such as `max-width: 1024px` (meaning “if the width is no greater than 1,024 CSS pixels”).

The possible operators between expressions are:

NOT operator

Using the `not` keyword before a conditional. `not` has a low precedence, meaning that it will always be applied to the whole expression unless we segment it using parentheses.

ONLY operator

Using the `only` keyword before a conditional. This means exclusivity. For example, `only screen` evaluates as true only on devices that don’t support any other type, such as `print`. Usually it’s a way to separate mobile devices from desktops, as desktop browsers support the `print` scheme.

AND operator

Using the `and` keyword between two conditions, such as `only screen and width: 320px`. We can use parentheses in conditions to avoid possible logical problems.

OR operator

There is no `or` keyword available; however, using a comma, we can emulate an OR operation, such as for orientation: `all and (orientation: landscape), all and (min-width: 700px)`.

Media features

In Chapter 10, we covered the pixel ratio extension to media queries (such as `-webkit-device-pixel-ratio`) and the discussion around compatibility to support different screens with different pixel densities. The following are the standard CSS media query features that we can use on mobile devices:

aspect-ratio

Defines the aspect ratio of the display area. The value must be two integers separated by a slash (horizontal/vertical), such as `aspect-ratio: 3/4`. This defines the relation between width and height, and it changes as the dimensions change. In mobile browsers, changing the device’s orientation usually changes `aspect-ratio`.

device-aspect-ratio

Defines the aspect ratio of the device. The value must be two integers separated by a slash (horizontal/vertical), such as `aspect-ratio: 4/3`. This defines the relation between `device-width` and `device-height`. On mobile devices it is the aspect ratio in portrait orientation.

device-width and device-height

Define the width or height of the device in the default orientation (portrait on mobile devices). The value is expressed in CSS pixels.



We can use `in` (inches) or `cm` (centimeters) in many media queries; however, these properties are not attached to real physical dimensions, but rather to a specific pixel relationship. Therefore, they are usually useless for making queries on physical dimensions.

width and height

Define the width or height of the current viewport. The value is expressed in CSS pixels. If we don't provide a viewport definition, the available width will be the default one (from 800 to 1,024px; see Chapter 6 for more information).

orientation

Defines the current device orientation, such as `landscape` or `portrait`. This is reevaluated automatically when the user changes the orientation.

resolution

Defines the current resolution (pixel density) of the device's screen. It accepts values in dots per inch (`dpi`) and dots per cm (`dpcm`). As we discussed in Chapter 10, this feature will accept dots per CSS pixel (`dppx`) in the near future. We should not rely on this property for every device, as some of them don't support it, and others just report a generic 96dpi resolution and not the real resolution. For today's devices, using the prefixed `device-pixel-ratio` is the solution.

Browser extensions

Mozilla accepts the `-moz-touch-enabled` Boolean attribute, which will be evaluated only on touch devices, and the `-moz-device-pixel-ratio` extension. WebKit-based browsers support `-webkit-device-pixel-ratio` and Opera (Presto engine) supports `-o-device-pixel-ratio`.



Firefox from version 16 and Opera from version 12 support resolution queries with `dppx`, such as `min-resolution: 1.5dppx` for high-resolution devices, to avoid the usage of the nonstandard and prefixed `device-pixel-ratio`.

all versus only screen

It's common in modern media queries to use the `only screen` type before any query instead of using `all`. The first big difference is that when we use `all`, our styles will also be applied when the page is printed. Besides that, the usage of `only screen` is preferred because starting the media definition with the `only` keyword hides the whole stylesheet from noncompatible browsers.



If a browser is not compatible with CSS3 media queries, it will understand CSS2 media queries, where only one word was used. So, if the first word of the media value is something that was valid in CSS2, such as `screen` or `all`, it will evaluate as true in these browsers, no matter what query follows it.

Useful queries

To define different styles for portrait and landscape, we can use the following declaration:

```
@media only screen and (orientation: portrait) {  
    // Portrait declarations  
}  
  
@media only screen and (orientation: landscape) {  
    // Landscape declarations  
}
```



The `orientation: portrait` and `orientation: landscape` declarations have been supported in Safari since iOS 3.2. It's a good idea to include both declarations in one stylesheet and not in separate files, so the browser won't need to make a new download when changing orientations.

If we are targeting only iPhone and iPod devices, to separate between original-size devices and larger devices (such as the iPhone 5) we can query on `device-aspect-ratio`:

```
@media only screen and (device-aspect-ratio: 2/3) {  
    // Original-size devices, such as iPhone 4S  
}  
  
@media only screen and (device-aspect-ratio: 40/71) {  
    // Larger devices, such as iPhone 5  
}
```

For iPhone and iPod devices in landscape mode, we can detect if the user has the full-screen option enabled (from version 6.0) using the following media query (with the

viewport width equal to device-width and no JavaScript fix for the viewport scaling bug running):

```
@media only screen and (orientation: landscape) and (height: 214px),
  only screen and (orientation: landscape) and (height: 181px) {
  // Full-screen mode with the default viewport
  // If you are changing viewport width, height values may change
}
```

For the aspect-ratio and device-aspect-ratio, we can apply any equivalent fraction, including the full width and height, such as 2/3, 4/6, or 320/480.



Remember that width and height are based on the current viewport; if we leave the default viewport, mobile devices will have the same dimensions as desktop windows. We can use device-width and device-height to rely on the device's dimensions regardless of the current viewport definition.

The Responsive Web Design approach is based on defining breakpoints on the current window width using a device-width viewport definition. The best solution is to start from the default values, then the smallest form, and add larger breakpoints one by one. For example:

```
// All default styles
@media only screen and (max-width: 480px) {
  // Smartphone-specific stylesheet
}
@media only screen and (min-width: 481px) {
  // Larger devices, tablets, and desktops
}
@media only screen and (min-width: 769px) {
  // Usually desktops
}
```

Another solution is to define all the groups as exclusive so they will not share any code, such as:

```
// All default styles
@media only screen and (max-width: 480px) {
  // Smartphone-specific stylesheet
}
@media only screen and (min-device-width: 481px) and
  (max-device-width: 768px){
  // Only tablets
}
@media only screen and (min-width: 769px) {
  // Usually desktops
}
```



Unfortunately, there is no media query to differentiate iPad 2 devices from iPad minis; they both have the same screen size in pixels and the same pixel ratio. While they have different screen densities, we don't have a way to query that.

CSS4 media queries

At the time of this writing, a working draft for CSS4 (<http://dev.w3.org/csswg/mediaqueries4>) is under discussion. It includes new attributes, such as:

script

A Boolean feature that means that JavaScript is available and enabled.

pointer

An enumeration that defines the presence and accuracy of a pointing device. The possible values are `none`, `coarse` for a pointing device with limited accuracy, and `fine` for an accurate pointing mechanism.

hover

A Boolean feature that means that hover events can be rendered on the device (such as `mouseover` or the `:hover` pseudoclass).

Internet Explorer 10 snap state

Windows 8 for desktops and tablets supports a Windows Store App mode for full-screen apps (previously known as Metro apps). These apps can work full-screen landscape, full-screen portrait, or “snapped.” The snap state, available on screens with a width of at least 1,366 pixels, allows us to have more than one app on the screen at the same time. The snapped app will have a width of 320 pixels.

Internet Explorer (the Windows Store version) can be also snapped. By default, IE10 will freeze the default viewport when the window is narrower than 1,024px. That means that if your IE window has a width of 320, 500, or 1,000px, the default viewport will remain at 1,024px, so your content will be scaled down. The same goes for IE10 in the snapped state with our website inside (as we can see in Figure 11-1).

To provide a different layout in IE10 when in snapped mode, you need to use media queries and use the Microsoft viewport extension that we discussed in Chapter 6. With this approach we can define that in windows narrower than 400 pixels we don't want a scaled-down viewport and we instead want a 320px viewport, as a mobile view:

```
@media screen and (max-width: 400px) {  
  @-ms-viewport {  
    width: 320px;  
  }  
  /* CSS for a 320px viewport in snapped mode goes here */  
}
```



Some e-ink book readers come with a web browser, such as the Amazon Kindle. They are the only monochrome web browsers out there today that can access our websites. Unfortunately, the Kindle browser evaluates itself as a color screen and not a monochrome one.



Figure 11-1. The Windows Store version of Internet Explorer in Windows 8 snap mode will narrow the viewport unless we override it using the viewport CSS extension.

If your website follows Responsive Web Design, you can use:

```
@-ms-viewport {  
  width: device-width;  
}
```

When we're defining the viewport to use in snap mode, the device-width constant is not really the device's width but the available width in snap mode, usually 320px.

When we're creating HTML5 Windows 8 apps—*not* websites for IE—Microsoft accepts two extensions to media query attributes: `-ms-view-state` and `-ms-high-contrast`. The first one accepts as values `fullscreen-portrait` and `fullscreen-landscape` when your app is the only app in the foreground, snapped when it's in snap mode, and `filled` when it is sharing the screen with another app in snap mode. The second attribute defines the current contrast mode on the system, as in `white-on-black` or `black-on-white`.

Selectors

The classic CSS 2.1 selectors are compatible with every device, and for the few that don't recognize them entirely, it may not be worth the effort to create alternatives. The mobile CSS-compatible selectors we can trust for every device are:

Universal

** (compatible but not recommended)*

Element

tagName

Class

.className

Unique ID

#elementId

Descendant

selectorselector

Child

selector > selector

Multiple

selector, selector

Pseudoclasses (link, visited, active, focus)

selector:pseudoclass

As we discussed previously, most mobile browsers today support CSS3. A compressed list of other selectors to use for these mobile browsers is:

Selector with attribute

selector[attribute]

Selector with attribute condition

selector[attribute<operator>value]



operator can be one of the following: equals (=), contains as one value (~=), begins with (^=), ends with (\$=), contains as a string (*=), or begins with and followed by hyphen (|=).

Negation

selector:not(selector)

Immediately preceded by

selector + selector

Preceded by
selector ~ selector

Pseudoclasses (after, before, root, nth-child(n), first-child, last-child, empty, and others)

selector:pseudoclass



Some CSS3 selectors don't work for mobile devices. How complex should a mobile website be, though? If it has that much complexity, perhaps we should consider simplifying.

If a browser has only partial support for some attribute or selector, that means the behavior is not complete. For example, the browser may not accept all the possible values, or it may render a selector properly in the original document but not apply the style if we change the DOM dynamically. This stylesheet fragment illustrates a noncritical use of CSS3 selectors:

```
input {  
    background-color: yellow;  
    border: 1px solid gray;  
}  
/* The next style will only work in CSS3-compatible browsers */  
input[type=button] {  
    background-color: silver;  
}
```

CSS Techniques

In this section we are going to talk about some well-known CSS techniques (reset CSS files, text formatting, and the box model) and see how the different browsers react to these features.

Reset CSS Files

It is very common in desktop web design to create a CSS hack to reset all the default margins and padding for common HTML elements. We can use this technique when developing for the mobile web, with some considerations: we should only reset the elements we are going to use, we should avoid the usage of the global selector (*) for performance purposes, and if we are using an external reset-CSS file we should consider merging it with our local CSS file.

Modern versions of reset files are available as part of the open source project *Normalize.css* (<http://necolas.github.com/normalize.css>), and are included in *HTML5 Boilerplate* (<http://html5boilerplate.com>).

Some browsers always create a margin around the whole document that cannot be deleted. And in the browsers that do allow you to delete the margin, remember that a zero margin may not be a good design decision.

Nokia also offers two markup and CSS templates for mobile web design for free through its developer site (<http://bit.ly/WpMswX>). Every template has a reset-CSS file. The following code is extracted from the mid-range device template, and we can adapt it to our needs:

```
html, body, div, span, object, blockquote, pre,
abbr, acronym, address, big, cite, code,
del, dfn, em, font, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center, dl, dt, dd, fieldset, form, label, legend,
caption, tr, th, td {
    margin: 0;
    padding: 0;
    border: 0;
    font-size: 100%;
    font-weight: normal;
    vertical-align: baseline;
    background: transparent;
}

p {
    border: 0;
    font-size: 100%;
    font-weight: normal;
    vertical-align: baseline;
    background: transparent;
}

a {
    margin: 0;
    padding: 0;
    font-weight: normal;
}

h1, h2, h3, h4, h5, h6 {
    margin: 0;
    padding: 0;
    border: 0;
    vertical-align: baseline;
    background: transparent;
}

body {
    line-height: inherit;
}

body table {
    margin: 0;
    padding: 0;
```

```

    font-size: 100%;
    font-weight: normal;
    vertical-align: baseline;
    background: transparent;
}

/* remember to highlight insertions somehow! */
ins {
    text-decoration: none;
}
del {
    text-decoration: line-through;
}

/* tables still need 'cellspacing="0"' in the markup */
body table {
    border-collapse: collapse;
    border-spacing: 0;
}

```



The Nokia Mobile Web Templates (<http://bit.ly/WpMswX>) are a set of templates (including XHTML and CSS files) for low-, mid-, and high-end devices that generate similar experiences across different devices, including hacks that solve some bugs, like the 100% width bug. They have been optimized for the Series 40 browser, Symbian browser, MeeGo browser, and Opera Mini.

Text Formatting

Text is the most common content type in a mobile website, and styling it in a way that maximizes compatibility can be a little tricky. Bold (`font-weight: bold`) and italics (`font-style: italic`) are reliably compatible, but support for other text-formatting features varies.

Font family

This will be our first problem in styling text for mobile browsers. There are no standards in terms of fonts for mobile operating systems, and most platforms have only one system font (generally sans serif).



NTT DoCoMo markup (for the Japanese market) still uses the old `font` tag for defining font properties like face, color, and size. Newer devices also support CSS. Other WebKit-based mobile browsers also support the `font` tag, but its use is not recommended. Use CSS instead.

We can provide specific font names (like Arial, Verdana, or Times New Roman) or generic font types (like serif, sans-serif, monospace, cursive, or fantasy).

For the best compatibility, you should use the default font and apply other attributes (color, size, etc.), unless you are on a known platform such as iOS or are defining your own custom fonts. If you want to define a font name, you should consider providing a list of alternatives. If the first font isn't available, the browser will try the second, then the third, and so on; if none of the listed fonts is available, it will use the default one.

Most browsers rely on the operating system for the available fonts, and apart from iOS—remember Steve Jobs's fanaticism about typography—almost every other mobile operating system has just one or a couple of fonts available.



When an operating system (like Android 2.x) has only a couple of proprietary font styles, even fonts that are safe on desktops, such as Arial or Times New Roman, will be rendered as the default one. Usually, these proprietary fonts are optimized for mobile screen reading.

Up to version 3.0, Android used a font called *Droid Sans*; a new font called *Roboto* replaced it in Android 4.0. Both are free fonts available from Google. For its devices, Nokia uses a *Nokia Sans* variation, and Microsoft uses *Segoe* for Windows Phone. BlackBerry 10 uses *Slate Pro* as the default font family, but it includes many others, such as Arial, Comic Sans, Browalia New, Georgia, Impact, Times New Roman, Trebuchet MS, and Verdana. For all these devices, leaving the default font or using `font-family: sans-serif` is enough.

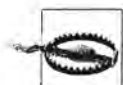
iOS has included several fonts from the beginning, and the list grows with every version. The website (<http://www.iosfonts.com>) has a complete list of the available fonts with a nice preview, and you can filter the list by operating system version, as we can see in Figure 11-2. Some of the fonts available on iOS are AmericanTypeWriter, Arial, Courier New, Georgia, Helvetica, Times New Roman, Trebuchet MS, Verdana, Zapfino, Marker, AppleSDGothicNeo, Avenir, Chalkboard, Futura, GillSans, HelveticaNeue, Optima, and Palatino. We can safely use these fonts on iOS, and provide alternatives for other devices.



Figure 11-2. Some operating systems support different font faces—iOS is the leader in this area, offering more than 60 system fonts.

CSS3 font-face

Before CSS3, some obscure techniques—including frameworks such as *sIFR* on Flash-enabled devices and *Cufón* for SVG-enabled devices—were used. CSS3 includes `@font-face` (<http://www.w3.org/TR/css3-fonts>), a custom font declaration that we can use on most modern smartphone and tablet platforms.



If you're delivering a custom font, you need authorization to distribute it. The font may be copyrighted, and you should make sure you have the right to distribute it as a custom font for your website. Google Web Fonts and Adobe Edge Web Fonts are all license-free. Adobe TypeKit allows you to use licensed fonts if you pay the royalties.

Delivering your own fonts means that the browser needs to download the font family file and store it temporarily in memory while your page is being rendered. That means you're adding a download overhead that will impact directly on your website's performance.

To create a font face definition, we use a new at-rule that will define a `src` attribute pointing to a source file for the font. For example:

```
@font-face {
  font-family: OReilly;
  src: url(http://myfonts.com/OReilly.ttf);
}
```

Then, any time we define `font-family: OReilly` in a CSS style, it will use that file. Font files define only one variant, so when we need a bold, italic, bold+italic, or any other weight variation, we need to define a different `@font-face` rule with the same font-family name, such as:

```
@font-face {  
  font-family: OReilly;  
  font-weight: bold;  
  src: url(http://myfonts.com/OReillyBold.ttf);  
}
```

The main problem with `font-face` is that the standard is format-agnostic. That means that we need to deal with format inconsistencies between platforms. Fortunately, we can define more than one format using comma syntax in the `src` style.

Google Web Fonts (<http://www.google.com/webfonts>) and Adobe Edge Web Fonts (<http://html.adobe.com/edge/webfonts>) are two projects that we can use for free (see Figure 11-3); they each provide hundreds of fonts. With these solutions, we just need to add one CSS link declaration or a script declaration. On the server side, their websites will deliver the right file format for the browser. We can use their servers as a content delivery network, getting the cache advantage, or download the files and host them ourselves.



Figure 11-3. With Google Web Fonts and Adobe Edge Web Fonts, we can use customized fonts in all modern tablet and smartphone browsers.

For example, to add the ABeeZee font from Google Web Fonts, we just need to add:

```
<link href='http://fonts.googleapis.com/css?family=ABeeZee'  
      rel='stylesheet'>
```

Remember that fonts are vector-based, so they will deliver great quality at every resolution. If you are not using dynamic content and are just using a custom font for your logo, you can compress your font file to just the characters you are going to use.



You can even create your own scalable icon gallery as a font file; Ico-Moon (<http://icomoon.io>) can help you with the task.

Font size

Which elements need a defined font size? For most cases, we should only define font sizes for headers and for element selectors (`h1`, `h2`, `p`, `div`).

We can use any measure for the font size, and almost every browser will understand it. However, it may not be rendered any differently. Only smartphone browsers with smart zoom support allow any font size to be rendered (like 13.5px in Safari on iOS).



For most of the mobile browsers, the best font size technique is to use relative constants (`xx-small`, `x-small`, `smaller`, `small`, `medium`, `large`, `larger`, `x-large`, `xx-large`) or relative units, such as `em`.

Operating systems have different font support. Some of them have only three possible sizes for text, and if we use the typical pixel definitions, two different sizes (for example, 12px and 14px) may be rendered identically. If we use relative constants (e.g., `large`), we have more probability of that text being rendered in a larger font. Another compatible way of specifying font sizes is to use `em` values. Using `em` values is perfect for supporting different screen sizes and DPIs because this unit is relative and scalable to the standard font in the device.

The default (`medium`) font size is generally the perfect size in the operating system for normal paragraph text, and for normal text we should leave it that way.

Text shadows

Another CSS3 feature is `text-shadow`. It allows us to define the color, *x*-offset, *y*-offset, and blur radius of a shadow to be applied to a text selector. For example, we can produce a shadowed headline with code like this:

```
h1 {  
  text-shadow: 0.1em 0.1em #AAA  
}
```

We can also define multiple shadow effects, using a comma to separate them.



If you're thinking about using this feature, remember that in the mobile world, the clearer the text is, the better for usability. Use text shadows with extreme care.

Text overflow

CSS3 adds a very useful feature for mobile web designs: text overflow. This property, available in some mobile browsers, allows us to specify that an ellipsis should appear at the end of a piece of text if it doesn't fit in its container in a single line, depending on the font and space available. This is great for reducing the amount of space taken up by links, and for previews or summaries that will be shown completely in a details page after the user clicks on them.

For example, we can show a title, and a truncated description with `text-overflow` set to `ellipsis`. When the user clicks on the title, via JavaScript we remove the `text-overflow` property, and the whole text of the description is shown. This maximizes the amount of content we can display on a page. This feature also works well on devices that support both landscape and portrait orientations: with text overflow we can assure the usage of only one line in both modes.

To use this feature, the paragraph (or other element containing the text) must have `overflow: hidden` set to avoid the continuing of the overflow text on the next line, `white-space: nowrap` to avoid wrapping, and some value for `text-overflow`.

The possible standard values for `text-overflow` are `clip` and `ellipsis`. The `ellipsis` value causes an ellipsis to appear after the last character that fits in the box (as shown in Figure 11-4, below). `clip` is the default value, which truncates the text without showing the ellipsis.



If you don't want to show the full text but you want to display more than just one line, there's no standard way to show the ellipsis at the end of the last line. Opera (and its mobile versions) supports the `text-overflow: -o-ellipsis-lastline` extension to solve that problem. On other browsers, however, we must rely on some large JavaScript frameworks, such as Three Dots (<http://www.jquerycode.com/theming/threedots>).

The BlackBerry 10 browser supports an extension (`text-overflow: -blackberry-fade`) that fades the text with an opacity gradient from left to right, as seen in Figure 11-4. Thanks to the fallback behavior of CSS3, we can first define `ellipsis` for all devices and then provide the `-blackberry-fade` declaration for BB10; all other browsers will ignore this.

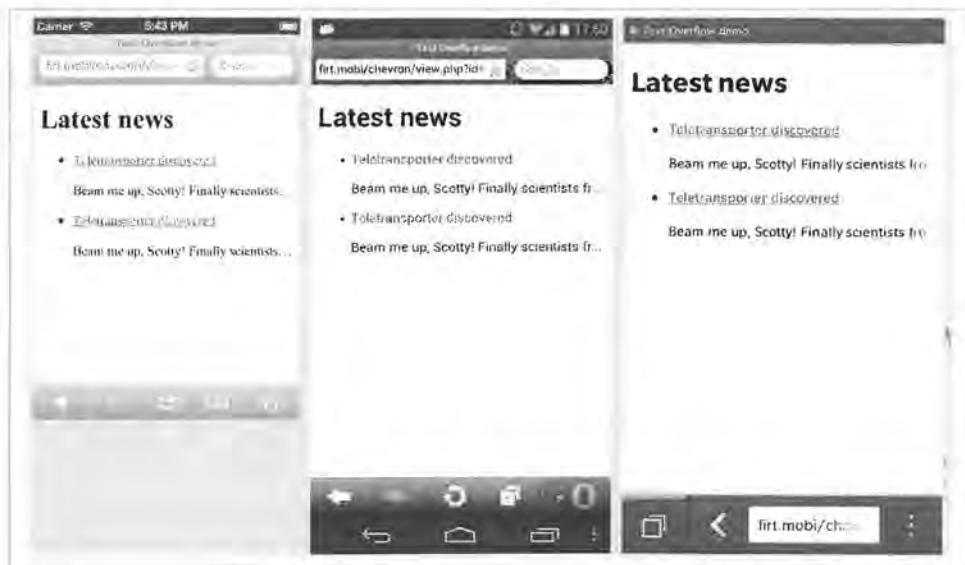


Figure 11-4. With text-overflow we can create nice effects for small screens (in this example, Safari on iOS, Opera Mobile for Android, and the BlackBerry 10 browser).

Here is a sample that produces the result shown in Figure 11-4:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Text Overflow demo</title>
<meta name="viewport" content="width=device-width">

<style type="text/css">

ul p {
  text-overflow: ellipsis;
  text-overflow: -blackberry-fade; /* For BB10 only */
  overflow: hidden;
  white-space: nowrap;
}

</style>
</head>
<body>

<h1>Latest news</h1>
```

```

<ul id="news">
  <li>
    <a href="#">Teletransporter discovered</a>
    <p>Beam me up, Scotty! Finally scientists from London have
      discovered teletransportation</p>
  </li>
  <li>
    <a href="#">Teletransporter discovered</a>
    <p>Beam me up, Scotty! Finally scientists from London have
      discovered teletransportation</p>
  </li>
</ul>

</body>
</html>

```

Text adjustment for small screens

Safari on iOS has created a CSS extension style especially for controlling the size of text, prepared for the zooming action: `-webkit-text-size-adjust`. Other WebKit-based browsers with other rendering engines have adopted the same property, such as Firefox (`-moz-text-size-adjust`) and Internet Explorer from version 9 (`-ms-text-size-adjust`).

This style accepts values of `auto` (the default), `none`, or a percentage (such as `200%`). By default, mobile browsers override a website's font sizes to allow the text to be read without any problems when the user zooms in on a paragraph. We can override this behavior with this style, turning it off (`none`) or defining a percentage zoom level to be applied to the default font size defined for the desktop website.

If we want to enhance a desktop website for mobile browsing, we should leave this style set to `auto`. However, if we want to override the default behavior or autosizing, we can use:

```

body {
  -webkit-text-size-adjust: none;
  -ms-text-size-adjust: none;
  -moz-text-size-adjust: none;
}

```

If a paragraph is prepared to be read in a desktop browser with a large viewport width, we can change this behavior using the `-webkit-text-size-adjust` attribute to enhance the mobile reading experience without changing the desktop appearance. Figure 11-5 shows the same text paragraph with this attribute set to `none` and to `400%`.

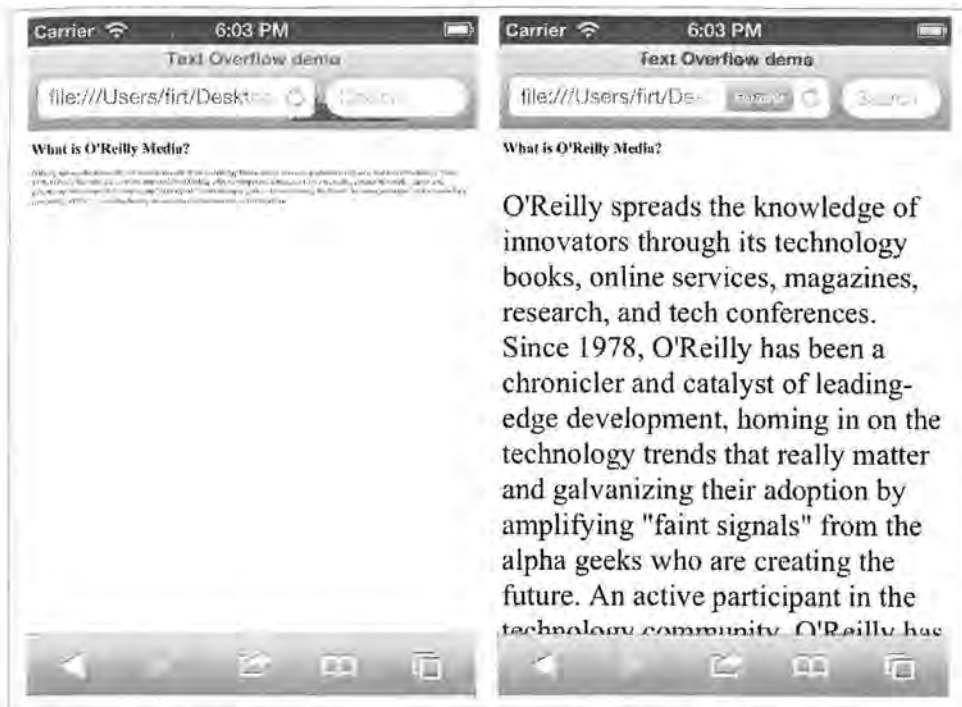


Figure 11-5. The same text paragraph with text adjustment off and with 400% as the value, in a website without any viewport definition—this feature is useful only in non-mobile web designs.

Text stroke and fill

The stroke and fill properties are a handy, nonstandard way of creating fancy effects in titles (with big fonts) without the use of images. At the time of this writing, only WebKit-based browsers support these attributes—that means Safari on iOS, the Android browser, Chrome, and the BlackBerry browser, using the `-webkit-` prefix.

For example:

```
<h1 style="-webkit-text-stroke: blue; -webkit-text-fill-color: yellow">
  Great Title!
</h1>
```

Multicolumn layout

More suitable for tablets or landscape phones, multicolumn layout (<http://www.w3.org/TR/css3-multicol>) is a new ability in CSS3 discussed in a separate standard. The feature is fairly self-explanatory: it allows us to define text and other content to be laid out in columns dynamically arranged based on two possible constraints, a fixed column width

or a fixed column count. Based on available length, the content will flow automatically between columns, following a newspaper or magazine layout.

Fortunately, all the tablet browsers support this, but at the time of this writing it is prefixed for WebKit-based browsers and Firefox. Internet Explorer 10 (for example, for the Microsoft Surface) and Opera support it without any prefix.

To create a multicolumn layout, we need to define `column-count` or `column-width`. If we define the count, the width will be calculated automatically, and vice versa. We can also optionally define a `column-gap` value that applies to the space between columns and a `column-rule` value (a border-type attribute that draws a line in the middle of the gap).

For example, if we want to apply a three-column layout to an article, as seen in Figure 11-6, we can use:

```
article {  
  -webkit-column-count: 3;  
  -moz-column-count: 3;  
  column-count: 3;  
  
  -webkit-column-rule: 1px dotted gray;  
  -moz-column-rule: 1px dotted gray;  
  column-rule: 1px dotted gray;  
}
```

Common Patterns

Even the most unique mobile web designs typically rely on a core set of common style patterns.

Display Properties

The standard display values (`none`, `block`, `inline`) are supported on almost all platforms, but in a limited way on older platforms, such as BlackBerry 4.x or 5.x.

There are also table display properties that I do not recommend using in mobile websites: `inline-table`, `table-column`, `table-cell`, and others. If we need a table, we can use the `table` element in HTML, and if we need a table-like layout, well, we should think twice about it.



Even when we're designing for some new smartphones—like the Samsung Galaxy SIII, which has a screen width of 720 pixels—we should avoid using tables and column layouts with more than two columns. The screen is still small, and we need to remember that it is a mobile device and think about the contexts in which it will be used.

The style display: none is used a lot in JavaScript and Ajax development and works without problems on modern mobile browsers.

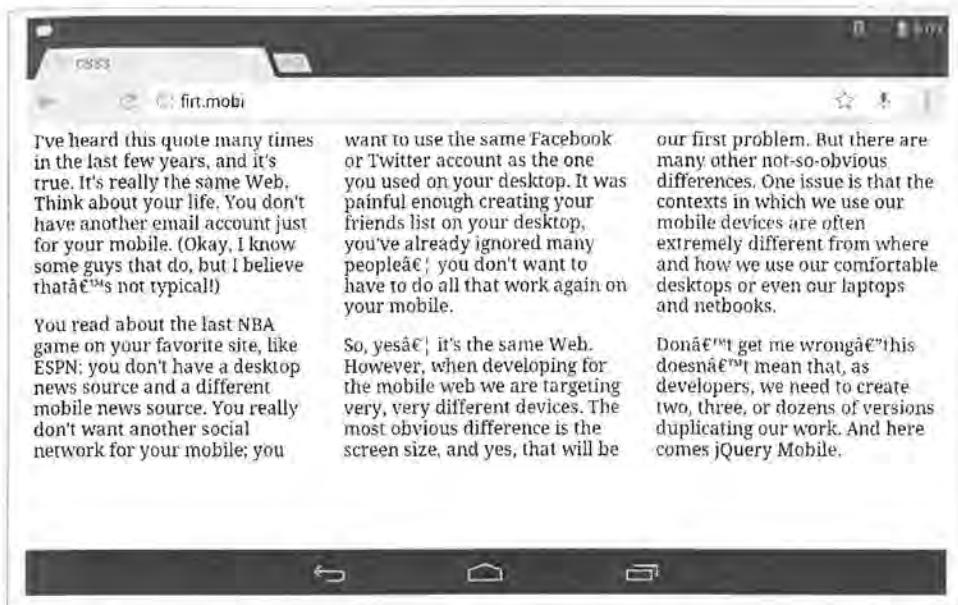


Figure 11-6. Multicolumn (magazine-like) layouts are mainly useful on tablets, because of the wider screen.

Absolute and floating positions

The standard position (`position: static`) is the most widely compatible and is recommended for mobile websites. This means that each element will be rendered in its normal position in the document.

Floating elements do work very well on most mobile devices. However, even on devices with average-sized screens it's best not to have more than two floating elements in the same row. This can be approached using `float: left` and `float: right`.

Relative positioning (`position: relative`) is trickier in mobile browsers. It defines movement (using `top`, `bottom`, `right`, `left`) from the original position as a static element.

The `clear` attribute (`clear: both`, for example) can be used after a floating element to ensure that no floating elements are allowed on the right, the left, or both sides of the element to which this attribute is applied.

Fixed position

To fix elements on the screen, in classic web development we use `position: fixed`, mixed with the properties `top`, `bottom`, `left`, and `right`. Until 2011, almost no mobile browser had support for fixed position, and it was downgraded to `position: absolute` instead. The main difference is that with absolute positioning the element scrolls with the page, so it's not an ideal solution if we want to have a fixed header or footer in our website.

Fixed position has been supported in Safari from iOS 5.0, in the Android browser since version 2.3, and in the BlackBerry browser since 7.0. The big problem that we still have is that `position: fixed` was prepared for a classic web environment and not for the more complex mobile scenario, where we have two different viewports (initial and current) and zoom abilities.

In compatible browsers, if we use `position: fixed` with a nonscalable viewport (`user-scalable=no`) and the viewport width set to `device-width`, it works great. However, when we allow zooming or if we are browsing a desktop-optimized website, the behavior can be really annoying, as you can see in Figure 11-7. Basically, the fixed elements are fixed to the current viewport and not the device/window viewport. Therefore, when the user zooms the page using the pinch gesture, the fixed elements are zoomed in too, but they remain fixed to the edges of the current viewport (which may or may not be visible on the screen after the zoom action).

If you have a desktop-optimized website with `position: fixed`, it's a good idea to remove that position value for mobile devices to avoid the problem that we can see in Figure 11-7. To do that we can use the following media queries:

```
@media only screen and (max-device-width: 400px) {  
  .fixedElement {  
    position: absolute; // We can fall back to absolute, hide  
                       // elements, or find another solution  
  }  
}
```



There is a JavaScript solution for creating a fixed element in noncompatible browsers, called iScroll (<http://www.mobiledxweb.com/iscroll>). The main problem with using this kind of JavaScript solution is that it impacts the performance of our websites and consumes battery power.

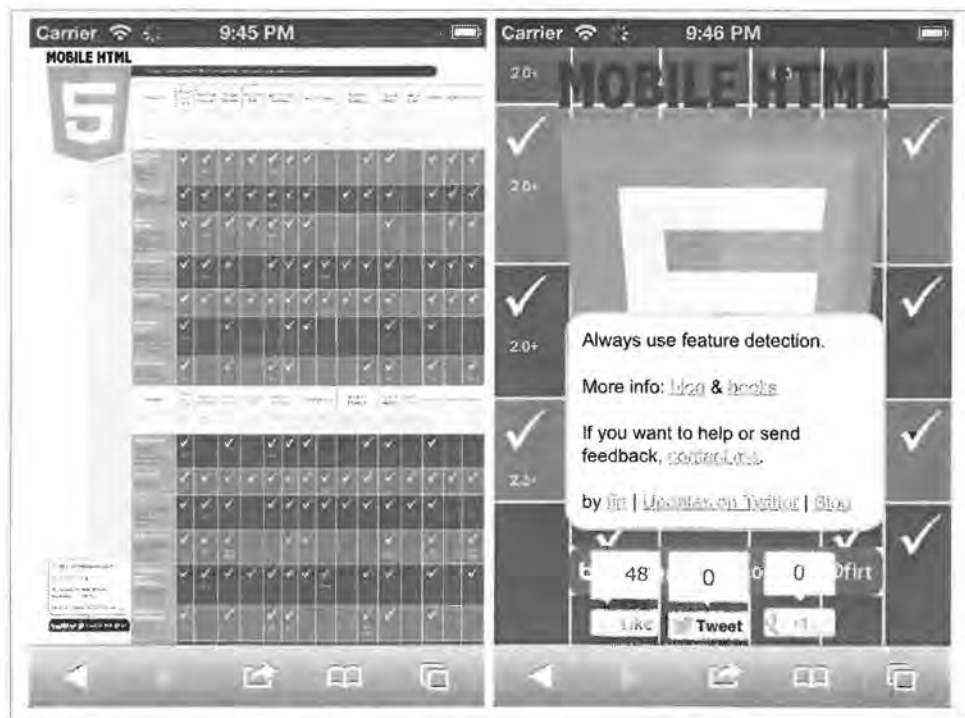


Figure 11-7. *position: fixed* when mixed with zoom-enabled viewports will lead to really strange behaviors when zooming in, as we can see in this example.

Modern layout techniques

Modern browsers support some CSS extensions for laying out elements on the screen. One technique is CSS Flexible Box Layout (<http://www.w3.org/TR/css3-flexbox>) (also known as Flexbox).



One of the biggest problems with Flexbox is that over the last few years the specification has changed from the original draft. Some browsers may still support the old spec, not the current and final specification, so you should use Flexbox with care.

When elements are defined as `display: flexbox` (for block elements) or `display: inline-flexbox` (for inline elements), all the children will float and fill the available space. Several CSS styles can be defined to customize the flexible layout, such as `flex-order`, `flex-pack`, `flex-wrap`, and `flex-direction`.

Opera Mobile is the only mobile browser at the time of this writing to support this feature without a prefix. Chrome and the BlackBerry 10 browser support it with the `-webkit-` prefix, while Firefox, Safari on iOS, and Internet Explorer 10 support their own prefixes, but with an old specification (depending on the version).

Microsoft has created a new layout mechanism that has been proposed as a standard, known as *CSS Grid Layout* (<http://www.w3.org/TR/css3-grid-layout>). At the time of this writing, only Internet Explorer 10 supports it; it involves the ability to design apps or content in a grid.

Rounded Corners

Designers seem to love rounded corners (shown in Figure 11-8), and for years this was the nightmare of every web developer who needed to lay out a box with this feature. Table-based layouts for rounded corners are inappropriate for the mobile web, so we can only rely on CSS solutions. If a device doesn't render the style, forget about rounded corners for that device. Any polyfill will just harm performance and isn't worth using on older devices.



The property for rounded corners is `border-radius`, and it was prefixed on older platforms, such as Safari for iOS before 4.0. If you have a considerable amount of users from those platforms, you can think about adding the prefixed versions too, such as `-webkit-border-radius` and `-moz-border-radius`.

The `border-radius` attribute can be defined as one value (like `5px` or `10%`), two values (top-bottom and left-right), or four values giving the radius of each corner separately. These are samples of different styles:

```
.rounded {  
    border-radius: 10px;  
}  
.rounded2 {  
    border-radius: 10px 20px;  
}  
.rounded3 {  
    border-radius: 3em 2em 3em 2em;  
}
```

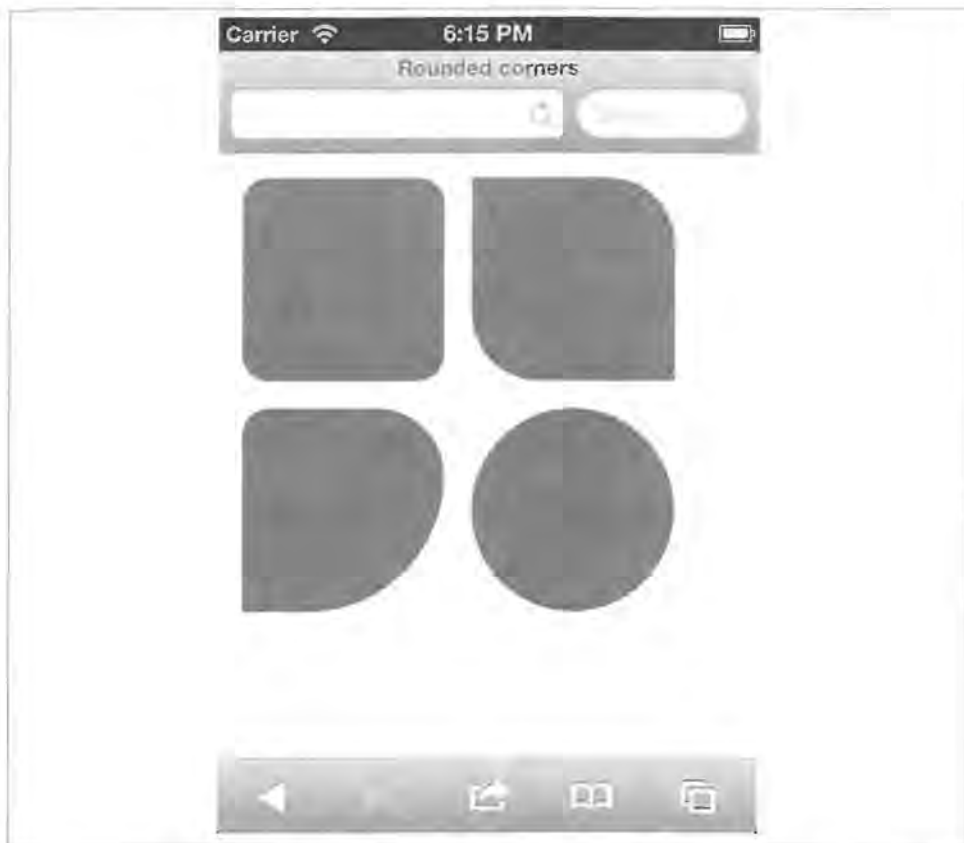


Figure 11-8. The rounded corners feature allows us to round any defined border or background color on compatible devices.

Border Image

The `border-image` CSS attribute is an excellent solution to the problem of creating a dynamically sized rectangle with custom borders. Its implementation is very similar to CSS sprites, and usage is simple. This technique is useful for buttons, titles, content zones, and every area where we want a custom border design without using tables.

The `border-image` attribute is part of the CSS3 standard, but at the time of this writing most mobile browsers still use prefixes (`-webkit-`, `-o-`, and `-moz-`). The exceptions are the BlackBerry browser from version 10 and Firefox from version 15; these no longer require a prefix for `border-image`. The most common syntax is:

```
border-image: url top right bottom left x_repeat y_repeat;
```

The `url` is the image location (or inline image), and the four edge values (`top`, `right`, `bottom`, `left`) are distance values measured from the image's sides. The center box

defined by the space not used by these four values will be used for the center pattern. For example, if we define 5 as the top, the box to which we are applying this style will have as the top border the top 5px of the border image.



The `border-image` property doesn't define the box's width and height or the border size; it is only used to define the contents of the border. If we need to change the dimensions of the box, we need to add `width` and `height` properties. We must also define the `border` property of the element, setting it to the desired size. The border image will be resized to the border size.

The `x_repeat` and `y_repeat` values are optional and can be defined as one of the following constants:

repeat

The portion of the image extracted using the *top* and *bottom* for `y_repeat` and using the *left* and *right* for `x_repeat` is repeated until it fills the available width/height of the box.

round

The image is repeated until it fills the available width/height of the box, but without any partial tile at the end; it is stretched so that it fits in the available space a whole number of times. This value has no effect in many mobile browsers.

stretch

The image is stretched to fill the entire width or height of the box without repetition.

The border image is cut into nine pieces, as we can see in the Figure 11-9. Four are used as corners and the others are used as background images for the sides and center.



If you are applying a border image to a button, it will not have any "pressed" effect. To create such an effect, you must change the active and/or focus pseudoclass, specifying another border image. Problems can occur when you try to change the way buttons are rendered dynamically, though, so for custom designs it is better to use links, or remove the default button rendering on WebKit-based browsers with `-webkit-appearance: none`.

The simplest way to define the border image is with the four values equal, using:

```
border-image: url distance;
```

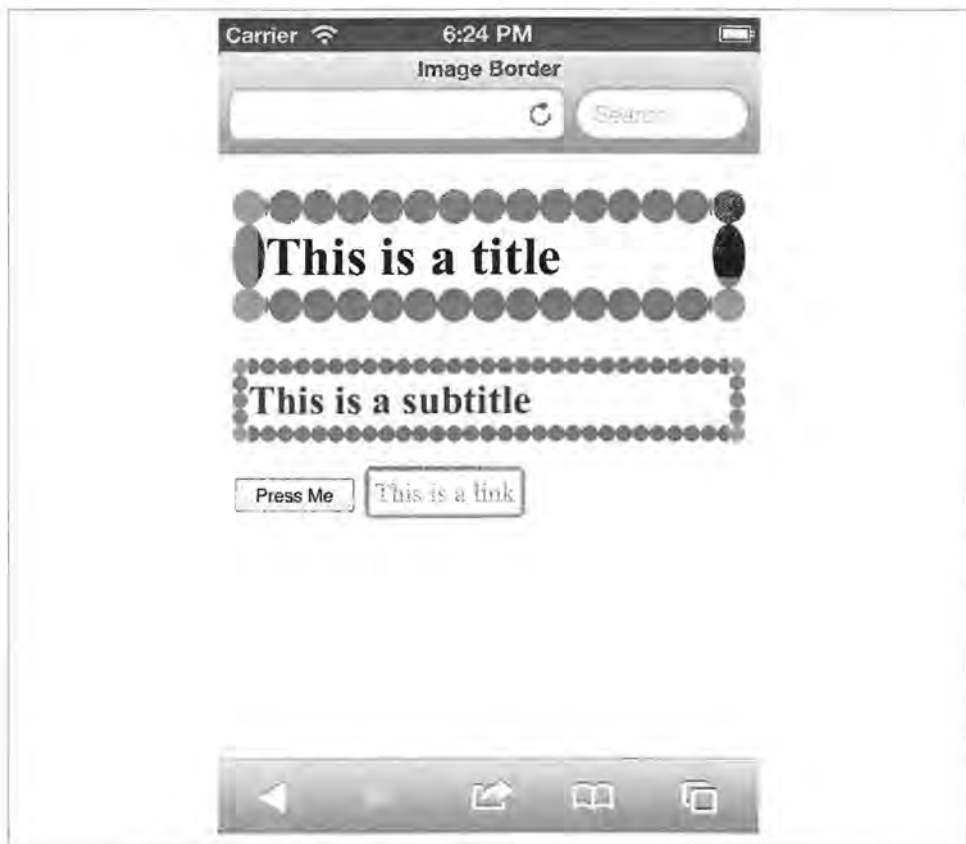


Figure 11-9. The image is cut into nine pieces and each one is used as either a corner or a part of the background.

This sample will produce the result shown in Figure 11-9:

```
<!DOCTYPE html>
<head>
<meta charset="UTF-8">
<title>Image Border</title>
<meta name="viewport" content="width=device-width">

<style>
/* We should use input[type=button] too, but for testing purposes
we will not use CSS3 */
input.bordered {
    border-radius: 10px;
    -webkit-border-image: url(border1.png) 6;
    -o-border-image: url(border1.png) 6;
    border-image: url(border1.png) 6;
}
```

```

a.bordered {
  -webkit-border-image: url(border1.png) 6;
  -o-border-image: url(border1.png) 6;
  border-image: url(border1.png) 6;
  color: white;
  text-decoration: none;
  padding: 3px;
}

h1 {
  border: 20px;
  -webkit-border-image: url(border2.png) 50 50 50 50 repeat stretch;
  -o-border-image: url(border2.png) 50 50 50 50 repeat stretch;
  border-image: url(border2.png) 50 50 50 50 repeat stretch;
}

/* The h2 will use the same image border but half size */
h2 {
  border: 10px;
  -webkit-border-image: url(border2.png) 50 50 50 50 round round;
  -o-border-image: url(border2.png) 50 50 50 50 round round;
  border-image: url(border2.png) 50 50 50 50 round round;
}
</style>
</head>

<body>
  <h1>This is a title</h1>
  <h2>This is a subtitle</h2>
  <input type="button" class="bordered" value="Press Me" />
  <!-- Safari applies border image to inline elements too -->
  <a href="http://mobilexweb.com" class="bordered">This is a link</a>
</body>
</html>

```

Another example is the implementation of the classic back button in iPhone user interfaces, using only left, right, and center zones (splitting the image into three parts):

```

<!DOCTYPE html>
<head>
  <meta charset="UTF-8">
  <title>Image Border</title>
  <style>
    #back {
      -webkit-border-image: url(border1.png) 0 5 0 15;
      -o-border-image: url(border1.png) 0 5 0 15;
      border-image: url(border1.png) 0 5 0 15;
    }
  </style>
</head>

```

```
<body>  
  <a href="/" id="back">Home Page</a>  
</body>  
</html>
```

Pseudoclasses

The pseudoclasses `link`, `active`, `focus`, and `visited` are compatible with all HTML browsers and standards. The question is, when do the pseudoclasses become active? Some situations are well known: for example, `link` is used for not-visited hyperlinks and `visited` is used if the links are in the previous browsing history.



In Chapter 8, we discussed the validation pseudoclasses for form elements, such as `required` and `invalid`.

What about the `focus` and `active` pseudoclasses, though? The behavior can vary in browsers with focus-based, cursor-based, and touch-based navigation. Some browsers, such as Safari on iOS, support the `active` pseudoclass on every element, while others support it only on anchor elements.

The well-known `hover` pseudoclass is not available in the WAP CSS standard, but it is compatible with most nontouch devices, assuming a similar behavior to `focus`. In touch devices, there isn't a `mouseover` event; the screen doesn't detect the finger or touch position until the user taps it (or clicks it).



There are a few specific exceptions, such as the Android-based Sony Xperia Sola and the Microsoft Surface tablet with Windows 8 Pro, both of which support `hover` states. In the case of the Surface tablet, `hover` events are triggered when using the stylus and not a finger.

Some mobile UIs for touch devices implement a two-tap pattern: if we tap once over an element, it will be like a `hover` effect, and if we tap again, it will be a click. This can be done with JavaScript and event handling.

Backgrounds

Changing the background color was the first new feature in XHTML MP that every WML developer used. It was heaven after the old black and white WML. Every mobile browser today understands the `background` property and its specific definitions, like `background-image` and `background-repeat`. However, we need to remember that on mobile devices, the context of the user can be very different from on a desktop. It is not

recommended to use a complex background, and it's best to use strongly contrasting foreground and background colors.



Some cloud-based browsers, such as Opera Mini and the Nokia Xpress browser, will ignore your background image if it's too large for their performance standards. Therefore, it's always a good idea to define a color as a backup for your image.

In compatible browsers, it will be very helpful to use data URI patterns for backgrounds to reduce network requests. One feature that can be buggy on mobile devices is the use of `background-attachment: fixed`. This allows the background image to be fixed even after scrolling.

CSS3 added the ability to specify multiple background images (separated by commas) and the `background-size`, `background-origin`, and `background-clip` properties.

Scrollable Areas

A common design pattern in desktop websites is to use a `div` (or other element) with a fixed size, and content larger than that size. Using the `overflow` property, we can define a value of `scroll`, `auto`, `visible` (the default), or `hidden` to indicate what to do with the content that is outside the bounds of the element. If we use a value of `scroll`, the `div` will have its own scroll bar on supporting devices. A similar technique involves the usage of the `iframe`.

For many years in the mobile web environment, scrolling in areas smaller than the whole viewport was discouraged, as it was an impossible mission for users on nontouch devices, and even on touch devices often there was no way to scroll inside them, or to do so the user needed a usually unknown gesture such as a double-finger scroll. This was a problem in Safari on iOS, up to version 4.3.

Since 2011, modern mobile browsers (including Safari on iOS since version 5) have added better support for scrolling small areas, removing the double-finger scroll gesture and adding some extensions to the `overflow` CSS property to match mobile-specific scrolling behaviors. The Android browser supports `overflow` from version 2.2.



For browsers that are not compatible with `overflow: scroll` or `overflow: auto`, some JavaScript frameworks have appeared offering a JavaScript-based solution. One example is Scrollability (<https://github.com/joehewitt/scrollability>).

The `overflow` extensions available in some browsers will generate a scroll with a nice native momentum style (deceleration and bouncing effects). For WebKit-based

browsers, compatibility started with Safari on iOS 5, and others followed. In these browsers, we can use `overflow: scroll` and the new `-webkit-overflow-scrolling: touch` extension. In the case of the BlackBerry 10 browser, to get a native scrolling feature we must use `-webkit-overflow: -blackberry-touch`.



Version 2.x of the Android browser has several issues with `overflow: scroll` elements.

Windows 8 Store apps created with HTML5 support a `-ms-overflow-style` property accepting the values `auto`, `none` (meaning no scroll bars will be visible, even though we can scroll), `scrollbar` (to show scroll bars all the time), and `-ms-autohiding-scrollbar` (to use overlay scroll bars that don't use up space and autohide after the interaction ends). We must use `overflow: scroll` at the same time as `-ms-overflow-style`.



When using `overflow: scroll` or `auto` we always need to define fixed dimensions, as in width or height. By default, if we leave automatic dimensions we won't have scrolling behavior and the box will just get the content size.

To have a touch or modern scroll bar area, we can then define:

```
.scrollable {  
  /* Always define a fixed width or height */  
  overflow: scroll;  
  -webkit-overflow-scrolling: touch;  
  -webkit-overflow-scrolling: -blackberry-touch;  
}
```

Content

The `content` attribute allows us to use the `after` and `before` pseudoclasses to define an image, some text, or an attribute value to be inserted after or before the selector.

Some browsers allow us to apply the `content` property to any selector, but this is not usually recommended because it will lead us to insert text and content in the CSS instead of the HTML document.

The following sample will add two stars after the link's text and a bullet image before:

```
a:after {  
  content: " ** "  
}  
a:before {  
  content: url('bullet.gif');
```

Opacity

Alpha transparency of elements inside a mobile web page will not work on many low-end and mid-range devices, so we should use it with care and be aware that it may not have a visible effect. The `opacity` CSS property allows the usage of a float number from 0 (totally transparent) to 1 (totally opaque). It applies to the element and all of its children.

In CSS3 we can also use the `rgba()` color function to make just the background or foreground colors semi-transparent, instead of the whole box and its contents—the first three parameters to this function represent the color and the fourth is the transparency value, again between 0 and 1. As it may be not compatible with non-CSS3 browsers, though, we should use a fallback mechanism, as in:

```
#myelement {  
  color: blue;  
  color: rgba(0, 0, 255, 0.5); // Blue with 0.5 opacity  
                                // on compatible browsers  
}
```

Cursor Management

CSS allows any web designer to define which mouse cursor should be used in any situation (generally, the body or a `:hover` selector). In the mobile world this is useful only for devices supporting cursor-based navigation, though, because they have the only browsers that show some kind of cursor over the screen.

The most useful cursors for mobile sites are `default`, `pointer`, and `progress`. The other cursors available (`resize` and `move`) can be very difficult to use in any mobile situation. We should use the `pointer` cursor for defining nonlink clickable zones (using a `:hover` selector), which may be handled by a JavaScript event function.

The `progress` cursor is often applied to the body dynamically with JavaScript to indicate to the user that a current operation is working. In browsers supporting focus and touch navigation, we should generate this pattern using a modal pop-up window with a floating loading image.

Modal Pop-up Windows

A modal pop-up is a floating `div` that displays important information to the user, while disabling and/or fading out the background content.

In the mobile world these are recommended only for smartphones and should be used with extreme care. If we are displaying only simple text, it is better to use the standard `window.alert` JavaScript function, as it will render properly on all devices.

Selection Management

Most smartphone and tablet browsers allow the user to select text and copy it to the clipboard, as seen in Figure 11-10. While this may be a desired feature in most situations, it can lead to some usability problems when we are managing touch events ourselves, or inside apps that will not provide useful content to copy.

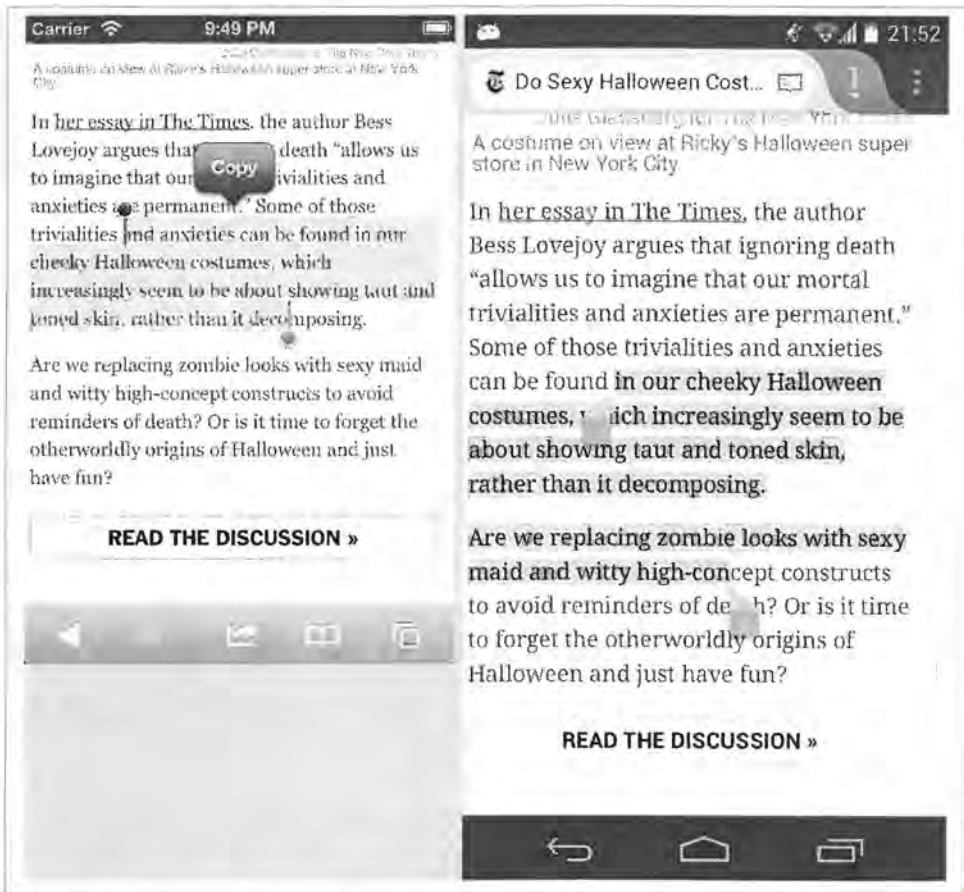


Figure 11-10. Some mobile browsers allow users to select text using some gesture, such as a double tap or two-finger tap over a paragraph, unless we deactivate the behavior using CSS extensions.

To remove the selectable behavior on some elements, we can use the not-yet-standard `user-select` property. This property was proposed in the CSS3 User Interface draft, but never made it to the full standard.

`user-select` accepts the following values: `none` (no content can be selected), `text` (the default; only text can be selected), `toggle` (the element's contents can be selected), `element` (only one child element at a time can be selected, such as a `p` element), `elements` (one or more elements can be selected), and `all` (only the contents as a whole can be selected).

At the time of this writing, this feature is being implemented with prefixes and not really following the standard possible values.

In WebKit-based browsers, such as Safari on iOS, the Android browser, and Chrome for Android, we can use `-webkit-user-select`, which allows `auto`, `none`, and `text`.

Firefox accepts the `-moz-user-select` variation, supporting the values `none`, `text`, `all`, `element`, and `-moz-none`. The difference between `none` and `-moz-none` is that the latter accepts that a child element can override the parent's `none` if it has a `-moz-user-select: text` attribute.

Internet Explorer 10 uses the `-ms-user-select` prefix and accepts `none`, `text`, `all`, `element`, and `auto`.



Opera with Presto engine and IE9 don't support `user-select` extensions, but they do support an HTML extension attribute that we can use for the same purpose: `unselectable="on"`.

To simplify, if we want to disable selection for compatible browsers we can use:

```
article {  
  -webkit-user-select: none;  
  -moz-user-select: none;  
  -ms-user-select: none;  
}
```



The CSS3 User Interface draft included a lot of other styles, such as `user-focus` and `toggle-group`, whose implementations are still quite strange on mobile browsers at the time of this writing.

Selection styles

If we want the user to be able to select content but want to customize the background color or style of the whole selection, we can use some nonstandard pseudoelements that are compatible with some browsers, such as `::selection`, which becomes `::-moz-selection` in Firefox. For example:

```
article {  
  -webkit-user-select: text;
```

```

-moz-user-select: text;
-ms-user-select: text;
}
article::selection {
  background-color: green;
  color: white;
}
article::-moz-selection {
  background-color: green;
  color: white;
}

```

In Figure 11-11, we can see the selection style being applied in some browsers.

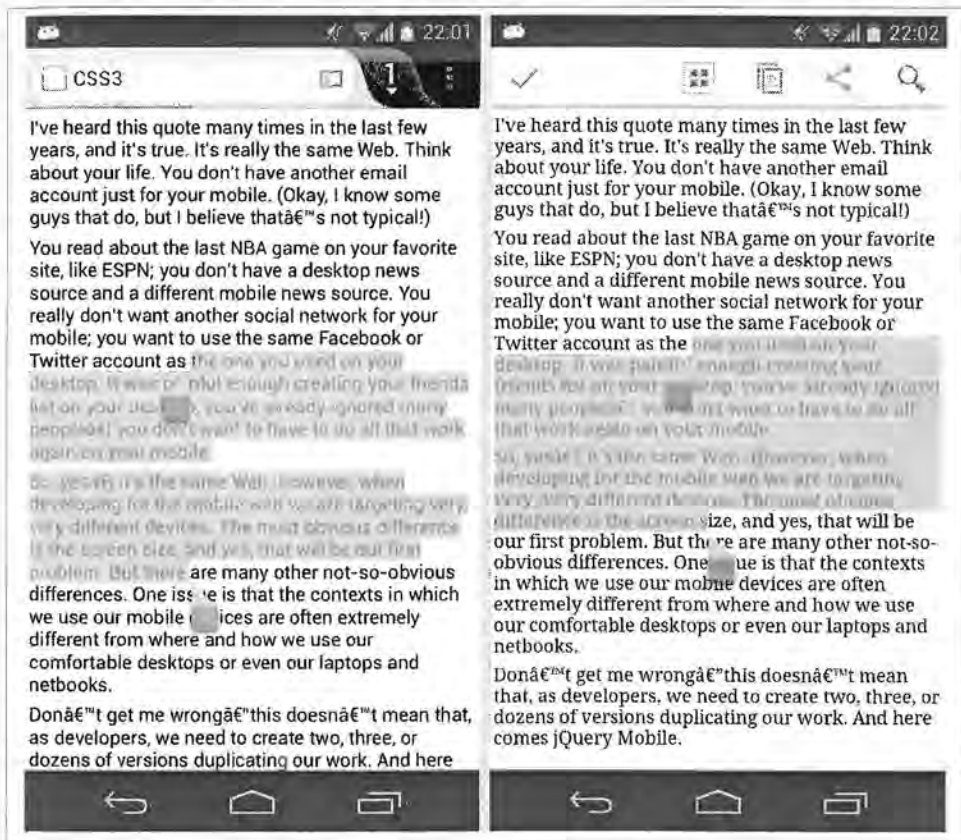


Figure 11-11. In some browsers we can customize how the selection pseudoelement is rendered, such as modifying the background color.

Touch Callout

When you use the long press or touch-and-hold gesture over a clickable area such as an element, a default callout appears, as we can see in Figure 11-12. In some situations, such as for a JavaScript-based hyperlink, a native web app, or a full-screen web app, disabling the callout may be a good idea. This is also a feature that we may want to disable if we are detecting that gesture in our own code.

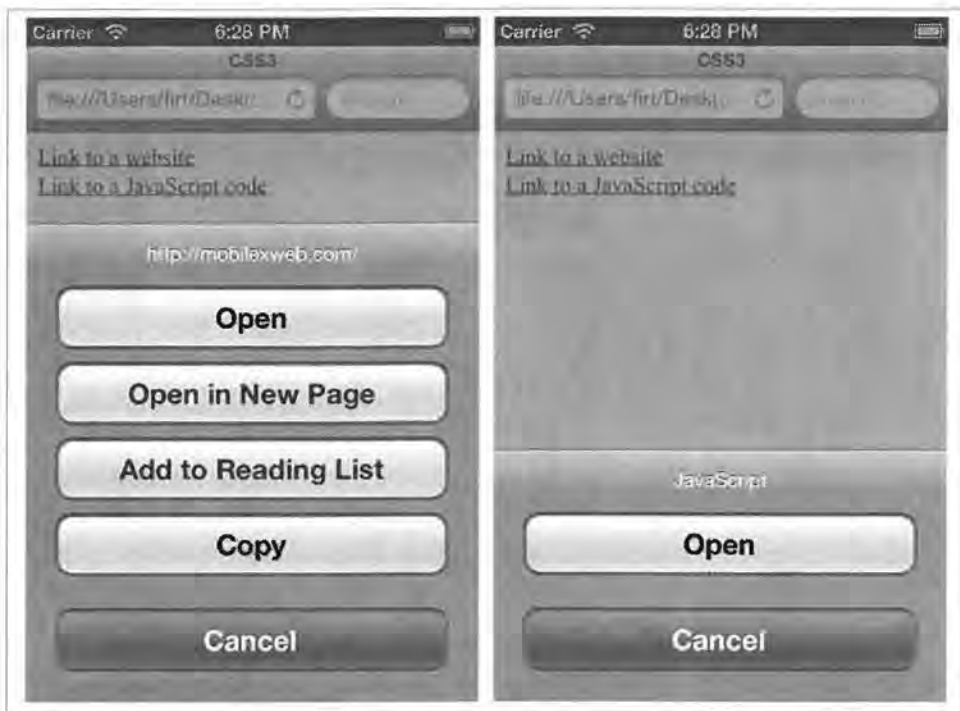


Figure 11-12. The callout is the pop-up window that appears when we touch and hold a clickable element, such as a hyperlink (even a JavaScript-based link).

Safari on iOS has added an extension, `-webkit-touch-callout: none`, that will disable that default pop-up window, as in:

```
a {  
  -webkit-touch-callout: none;  
}
```

To remove callouts only for JavaScript-based hyperlinks, we can use a CSS3 “starts with” attribute selector:

```
a[href^=javascript] {  
  -webkit-touch-callout: none;  
}
```

Other WebKit browsers, such as Google Chrome, currently ignore this attribute, and there are no alternatives for other browsers at the time of this writing.

Highlight Color

When a mobile user taps on a clickable element, such as a link, the default highlight color is shown as a layer of color over the element with a default alpha style. We can override the color or remove the highlight effect using the extension `-webkit-tap-highlight-color`.

If we want to define our own color, we should always make it transparent, with the `rgba()` function. If we provide an opaque color, such as yellow or #CCC, the clickable element will be totally obscured by that color.

With the `rgba()` function we can override the default alpha value and apply a 0 value (fully transparent) to remove the effect completely, or a 1 value (fully opaque) to obscure the element completely:

```
a.noeffect {  
  -webkit-tap-highlight-color: rgba(0, 0, 0, 0);  
}
```

This property works on most WebKit-based mobile browsers, such as Safari on iOS, the Android browser, and Chrome.



Remember that you can use some pseudoclasses, such as `active`, to change an element's properties when the user taps on that element.

Appearance Override

WebKit-based browsers and Firefox allow us to override the default appearance of an element with the `-webkit-appearance` and `-moz-appearance` properties. For example, we can remove the default platform-specific styling of a button or a text input using the `none` attribute, as follows:

```
button {  
  -webkit-appearance: none;  
  -moz-appearance: none;  
}
```

If we use one of the available values, we can change the appearance of the element so it resembles a native control in the operating system. A partial list of possible values includes: `button`, `button-bevel`, `checkbox`, `default-button`, `listbox`, `listitem`, `media-fullscreen-button`, `media-mute-button`, `media-play-button`, `radio`, `searchfield`,

searchfield-cancel-button, slider-horizontal, slider-vertical, square-button, textarea, and textfield.



WebKit-based browsers support the `-webkit-text-security` extension that defines the character to display in password fields for each character the user enters. Possible values are `circle`, `disc`, `none`, and `square`.

CSS Sprites

Using CSS sprites is a great modern web design technique for reducing the number of image server requests on a web page. There are a lot of online resources and books available on this technique. For now, suffice it to say that if you have many images in your site (preferred logos, icons, background images, flags, and so on), you can reduce all of those to one big image with all the originals inside and use a CSS mask to determine which portion of it to show in each container.

This technique has a great impact on web performance, but for mobile applications we should think twice before using it and analyze the possible problems. First, we need full `background-position` CSS property compatibility (the mobile standards include this, so it's not really an issue). The second consideration is that we will not be using `img` tags. In their place, we will use any block element (`div`) or any block-converted element using `display: block`, such as a `span` or a `tag`. This means that we cannot provide alternative text for the images, and the browser won't know how much space to allocate for each image until it renders the CSS file.

Finally, in some browsers this technique can have an impact on rendering performance, because the big image will be duplicated in memory for each usage. We need to balance the performance gained through the reduction of requests with the performance lost in the rendering engine in some browsers.

Samples and Compatibility

Let's create a sample using two techniques: using an original block element (`div`) and using an original inline element (`a`) converted to a block element.

The original document without CSS sprites is the following country list:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>CSS Sprites Demo</title>
<style type="text/css">
```

```

ul {
    list-style: circle;
}

ul li {
    padding: 0px;
    margin-bottom: 5px;
}

ul li img {
    margin: 0px 10px 0px 0px;
    vertical-align: middle;
    border: 1px solid gray;
}

</style>
</head>

<body>
<h1>The Best Seller</h1>
<h2>Select your nearest country</h2>
<ul>
  <li><img src='ar.png' width='30' height='19' alt='AR'>
    <a href='ar'>Argentina</a></li>
  <li><img src='br.png' width='30' height='19' alt='BR'>
    <a href='br'>Brazil</a></li>
  <li><img src='es.png' width='30' height='19' alt='ES'>
    <a href='es'>Spain</a></li>
  <li><img src='fi.png' width='30' height='19' alt='FI'>
    <a href='fi'>Finland</a></li>
  <li><img src='jp.png' width='30' height='19' alt='JP'>
    <a href='jp'>Japan</a></li>
  <li><img src='us.png' width='30' height='19' alt='US'>
    <a href='us'>United States</a></li>
</ul>
</body>
</html>

```

The previous sample uses six images that can be converted into a single one (saving five requests and the HTTP and PNG headers) with the following code:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>CSS Sprites Demo</title>
<style type="text/css">

  ul {
    list-style: circle;
  }

  ul li {

```

```

padding: 0px;
margin-bottom: 5px;
}

ul li div {
    /* We define the large image as a background for all divs
       that represent an image*/
    background:url(sprite.png);
    width: 30px;
    height: 19px;
    float: left;
    border: 1px solid gray;
    margin-right: 10px;
}

</style>
</head>

<body>
<h1>The Best Seller</h1>
<h2>Select your nearest country</h2>
<ul>
    <li><div style="background-position: 0px 0px;"></div>
        <a href='ar'>Argentina</a></li>
    <li><div style="background-position: 0px -29px;"></div>
        <a href='br'>Brazil</a></li>
    <li><div style="background-position: 0px -58px;"></div>
        <a href='fi'>Finland</a></li>
    <li><div style="background-position: 0px -87px;"></div>
        <a href='jp'>Japan</a></li>
    <li><div style="background-position: 0px -116px;"></div>
        <a href='es'>Spain</a></li>
    <li><div style="background-position: 0px -145px;"></div>
        <a href='us'>United States</a></li>
</ul>
</body>
</html>

```

This produces the result shown in Figure 11-13.

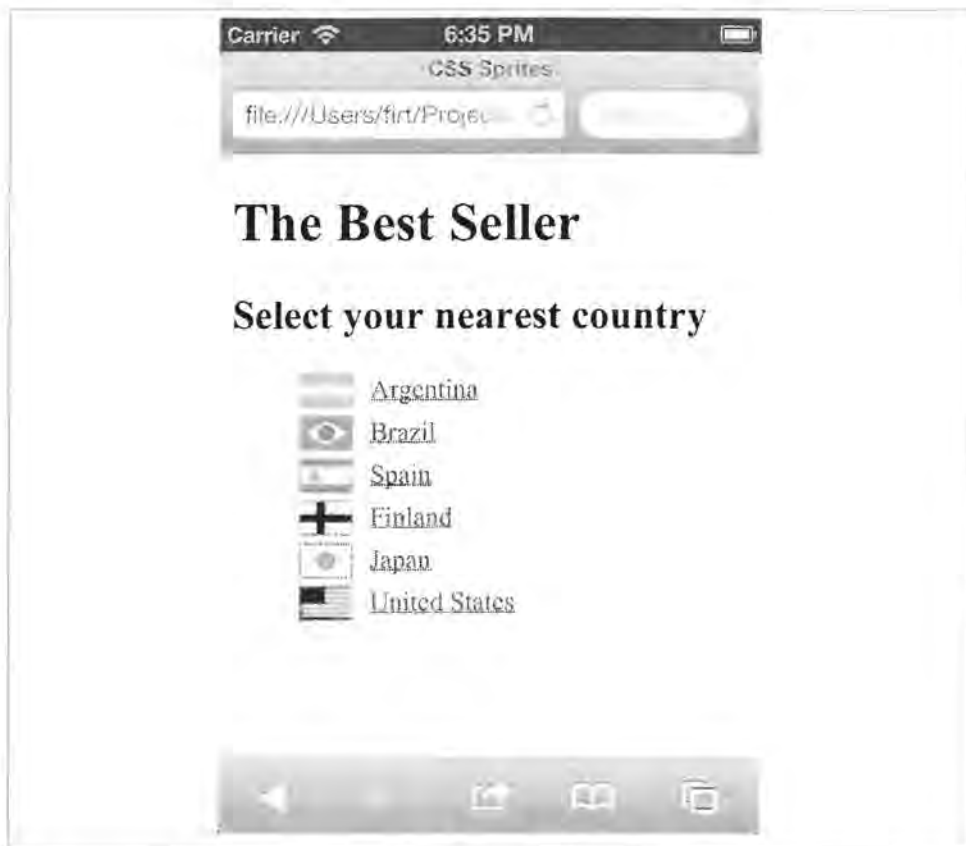


Figure 11-13. In compatible browsers, using CSS sprites produces the same result using a single image as using six separate images for the flags.



There are plenty of online CSS sprites generators where you can upload all your images and receive in seconds one big image and the CSS code to replace each of the original `img` tags. Examples include Spritegen (<http://spritegen.website-performance.org>) and CSS Sprites (<http://csssprites.com>).

Now let's look at applying the same technique to a nonoriginal block element, such as the `a` tag. The only problem will be the flag border: as we use the same `a` tag for the image and the text, we cannot define a border. The code looks like this:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
```

```

<title>CSS Sprites demo</title>
<style type="text/css">

    ul {
        list-style: circle;
    }

    ul li {
        padding: 0px;
        margin-bottom: 5px;
    }

    ul li a {
        /* We define the large image as a background for all divs
           that represent an image */
        background:url(sprite.png);
        /* We need to create block elements */
        display: block;
        /* We need the background to not be repeated */
        background-repeat: no-repeat;
        height: 19px;
        padding-left: 40px;
    }

</style>
</head>

<body>
<h1>The Best Seller</h1>
<h2>Select your nearest country</h2>
<ul>
    <li>
        <a href='ar' style='background-position: 0px 0px;'>Argentina</a>
    </li>
    <li>
        <a href='br' style='background-position: 0px -29px;'>Brazil</a>
    </li>
    <li>
        <a href='es' style='background-position: 0px -116px;'>Spain</a>
    </li>
    <li>
        <a href='fi' style='background-position: 0px -58px;'>Finland</a>
    </li>
    <li>
        <a href='jp' style='background-position: 0px -87px;'>Japan</a>
    </li>
    <li>
        <a href='us' style='background-position: 0px -145px;'>
            United States
        </a>
    </li>
</ul>

```

```
</body>  
</html>
```



Using CSS sprites is not recommended for big files or photo images. If you are using PNG images, the best way to approach it is to group icons with a consistent color palette.

CSS Sprites Alternatives

The idea of minimizing the number of requests to the server is very interesting, even if you reject the usage of CSS sprites. That is why we need to think about alternatives to this technique for some specific situations.



Image maps are the first technique that comes to mind as a CSS sprites alternative. However, they are not recommended for nontouch navigation, because image maps in nontouch devices can have a negative impact on usability.

Inline images

As we discussed in the last chapter, inline images are a great technique for compatible browsers. When designing for browsers that understand them, we can copy the first sample (the original document without CSS sprites) and replace the URL of each image with the `data:` representation.

Join images

If the images are near one another horizontally or vertically, as in our sample, we can consider joining all the images into one. The concept is similar to CSS sprites, but we set up the image as a single-use background, adjusting the margins and padding so that the elements are properly aligned with the different parts of the image. This technique can have poor results on old devices with limited support for margins and padding.



If we use the original code but define a good cache policy on the server, subsequent pages of the site will load faster than if we used CSS sprites, because no rendering work will be required.

CSS3 Modules

Besides the standard CSS specification, there are different modules that a browser may or may not support. In this section we will cover the most useful modules that we can use today in mobile browsers.

Gradients

The CSS3 Image Values (<http://www.w3.org/TR/css3-images>) specification adds a way to define gradients as image functions inside any style that needs an image as a value. Instead of using the `url()` function to provide the URL of the image, we can use the available gradient functions, such as `linear-gradient()`, `radial-gradient()`, `repeating-linear-gradient()`, and `repeating-radial-gradient()`. This technique enables us to create really nice backgrounds for titles, containers, and cells with minimal code.

The specs have changed over time, and compatibility is mixed between old spec, and new spec prefixed and unprefixed. Safari on iOS 3.0 was one of the first browsers to support this feature and we can use it today on most modern mobile browsers, including Safari from iOS 3, the Android browser from 2.1, Chrome, Firefox, the BlackBerry browser from 7.0, Opera from 11.1, and Internet Explorer from version 10.



For browsers that do not support CSS gradients, such as Internet Explorer 9 for Windows Phone or BlackBerry 5, using a plain color instead of a background image will result in better performance.

Internet Explorer from version 10, Opera from 12.1, and Firefox from 16 don't need prefixes anymore, so we can use the standard code. For all the others we must use the `-webkit-`, `-moz-`, and `-o-` prefixes.



Repeating gradients have appeared late in mobile browsers—Safari supports them from iOS 5.0, the Android browser from 4.0, and the BlackBerry browser from BB10.

While gradients are beyond the scope of this book, there are some websites that will allow you to design and preview your gradients with a visual tool and get the cross-platform code pretty easily. Some tools allow you to upload a bitmap file with a gradient inside and export it to CSS gradients. Useful online tools include:

- Ultimate CSS Gradient Generator (<http://www.colorzilla.com/gradient-editor>) (Figure 11-14)
- Generate It Gradient (<http://www.generateit.net/gradient>)
- CSS3 Gradient Generator (<http://css3gen.com/gradient-generator>)

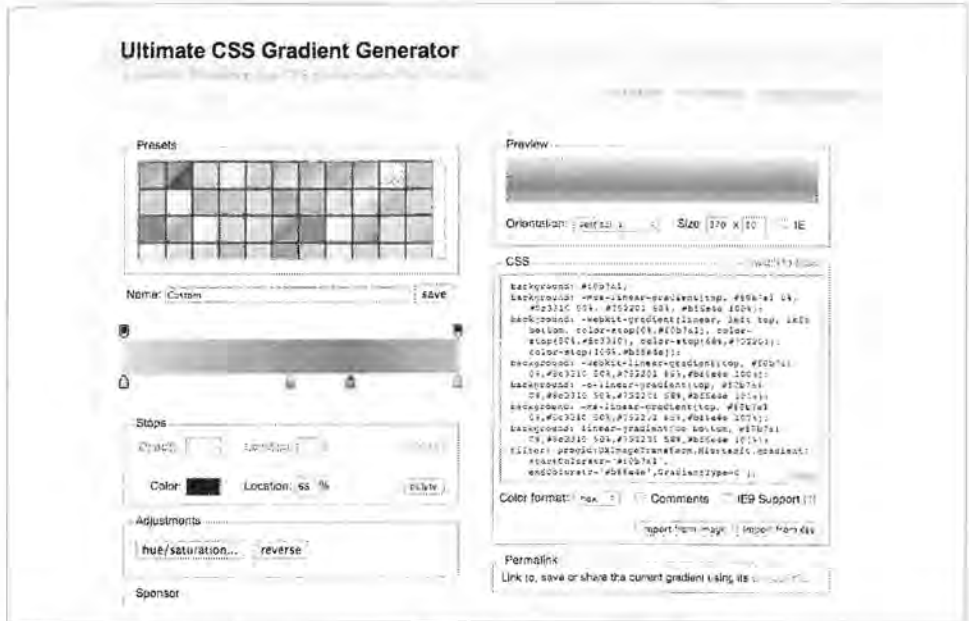


Figure 11-14. With the Ultimate CSS Gradient Generator, we can design our gradient in a visual drag-and-drop editor and get the CSS standard code ready to paste.



Older versions of WebKit-based browsers support only an old specification for gradients, using the `-webkit-gradient()` function and defining linear or radial as the first parameter. The final spec has different functions for each type.

Just to give you a quick sample, the following code will show you a linear gradient with a fallback plain color for list items:

```
li {
  background: #f0f9ff;
  background: -moz-linear-gradient(top, #f0f9ff 0%, #cbebff 47%,
    #a1dbff 100%);
  background: -webkit-gradient(linear, left top, left bottom,
    color-stop(0%,#f0f9ff), color-stop(47%,#cbebff),
    color-stop(100%,#a1dbff));
  background: -webkit-linear-gradient(top, #f0f9ff 0%, #cbebff 47%,
    #a1dbff 100%);
  background: -o-linear-gradient(top, #f0f9ff 0%, #cbebff 47%,
    #a1dbff 100%);
  background: linear-gradient(bottom, #f0f9ff 0%, #cbebff 47%,
    #a1dbff 100%);
}
```

In Figure 11-15, we can see different gradients in action, including linear and radial.

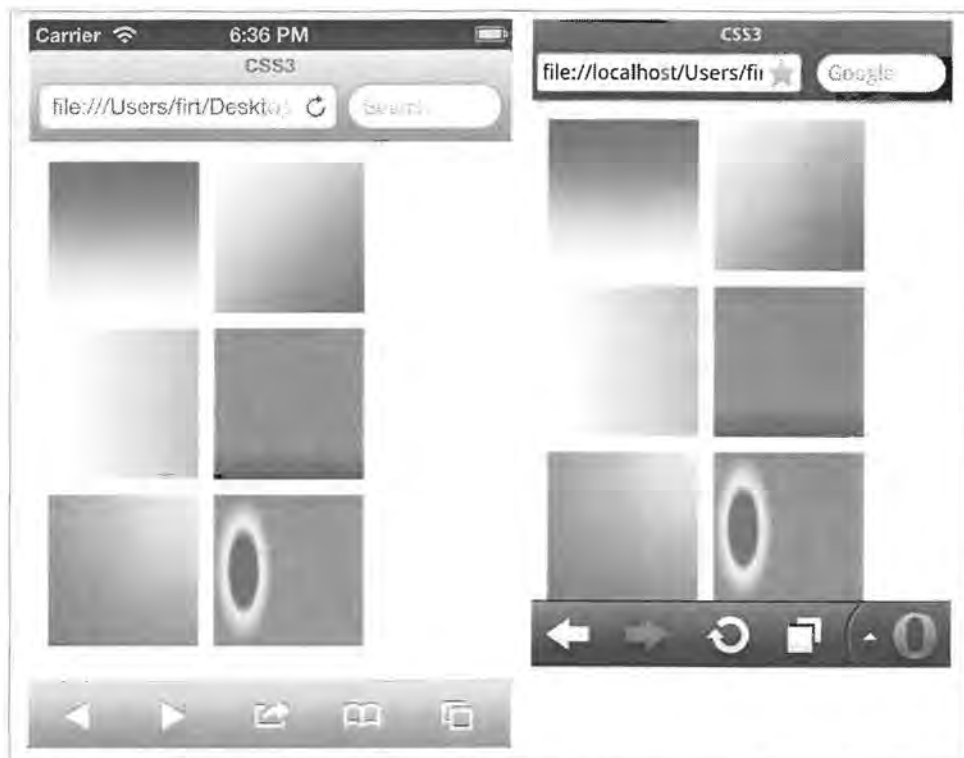


Figure 11-15. With just CSS you can create different gradient effects for most smartphones and tablets without using images.



As of version 6.0, Mobile Internet Explorer supports filters and transitions, using CSS extensions with the `filter` style. You can create alpha, chroma, shadow, glow, mask, and other effects. For more information, see www.mobilexweb.com/go/iefilter.

Reflection

Reflection or mirror effects are available as an extension for WebKit browsers, but there is no W3C standard behind them. They can be used on any content, including images. Remember, though, that we are designing for mobile screens and we don't want to waste too much space.



The reflection image doesn't change the layout or the size of the element's original content box. It is only part of the container's overflow.

To create a reflection effect in WebKit-based browsers such as Safari on iOS, use the `-webkit-box-reflect` attribute with the following syntax:

```
-webkit-box-reflect: direction offset [mask-box-image];
```

The *direction* can be *above*, *below*, *left*, or *right*; the *offset* is the distance (in px or %) from the original element at which the reflection should appear; and the optional *mask-box-image* is generally a gradient function that will work as a mask for the reflection image. If no mask image is defined, a normal mirror will be used.



CSS reflection works in Safari on iOS, the Android browser, Chrome, and BlackBerry from 7.0.

The type of reflection effect typically seen in Web 2.0 websites has the following attribute values, as we can see in Figure 11-16:

```
-webkit-box-reflect: below 3px -webkit-linear-gradient(top, transparent 0%,  
                    rgba(255, 255, 255, 0.5) 50%, white 100%);
```



Figure 11-16. CSS reflection is a nonstandard effect that will not affect the element's original dimensions.

Masks

Like reflections, masks are a WebKit-only extension that gives us access to a graphic design feature that has been missing for years in web development: masked images. We can use a masked image to apply any regular or irregular crop to the original image, or, if using an alpha mask (or even a gradient function), to create a really nice visual effect over any image, like a fuzzy border. The mask properties are analogous to the background properties. For applying a mask, we have a shortcut property, `-webkit-mask`, and specific properties for the position.

The syntax of the shortcut version with all the optional parameters is:

```
-webkit-mask: attachment, clip, origin, image, repeat, composite,  
             box-image;
```

Of course, we also have access to all the properties separately, as in `-webkit-mask-attachment`, `-webkit-mask-clip`, and so on. There are a lot of possibilities, but typically an image (alpha or not, PNG or SVG) or a gradient function is used as the image value. For example:

```



```

Transforms

At the time of this writing, CSS transforms (<http://www.w3.org/TR/css3-2d-transforms>) is a draft standard but is quite safe to use to apply functions to any element to generate visual effects without using images, canvas, or SVG. The transformation options work in many browsers and include a main shortcut transform property, a couple of CSS functions, and some extension attributes. As with many extensions, some browsers are still using prefixed versions at the time of this writing.

The usage is very simple: we use the CSS property transform, applying as a value any of the CSS functions listed in Table 11-1.

Table 11-1. CSS transform functions available in CSS3

Function	Description
<code>matrix(m11, m12, m21, m22, tX, tY)</code>	Specifies a matrix transformation of six values with two translation elements.
<code>matrix3d(m00, m01, m02, m03, m10, m11, m12, m13, m20, m21, m22, m23, m30, m31, m32, m33)</code>	Specifies a 3D matrix transformation of 4x4.
<code>perspective(depth)</code>	Maps a viewing cube onto a pyramid whose base is far away from the viewer. This property is not part of the standard and is available only in WebKit-based browsers.
<code>rotate(angle)</code>	Defines a 2D rotation around the origin of the element.
<code>rotate3d(x, y, z, angle)</code>	Defines a 3D rotation with [x,y,z] as the direction vector of the rotation.
<code>rotateX(angle)</code>	Specifies a clockwise rotation around the x-axis.
<code>rotateY(angle)</code>	Specifies a clockwise rotation around the y-axis.
<code>rotateZ(angle)</code>	Specifies a clockwise rotation around the z-axis.
<code>scale(scaleX, [scaleY])</code>	Performs a 2D scale operation.
<code>scale3d(scaleX, scaleY, scaleZ)</code>	Performs a 3D scale operation.
<code>scaleX(value)</code>	Scales along the x-axis.
<code>scaleY(value)</code>	Scales along the y-axis.
<code>scaleZ(value)</code>	Scales along the z-axis.

Function	Description
<code>skewX(<i>angle</i>)</code>	Performs a skew transformation around the x-axis.
<code>skewY(<i>angle</i>)</code>	Performs a skew transformation around the y-axis.
<code>translate(<i>deltaX</i>, <i>deltaY</i>)</code>	Specifies a 2D translation vector.
<code>translate3d(<i>deltaX</i>, <i>deltaY</i>, <i>deltaZ</i>)</code>	Specifies a 3D translation vector.
<code>translateX(<i>value</i>)</code>	Performs a translation around the x-axis.
<code>translateY(<i>value</i>)</code>	Performs a translation around the y-axis.
<code>translateZ(<i>value</i>)</code>	Performs a translation around the z-axis.



Rotation transforms, such as `rotate` and `skew`, need new kinds of units for defining values. CSS defines `deg` as degrees (`180deg`), `rad` as radians (`1rad`), and `turn` as turns (`0.5turn`).

We can change the origin point of the transformation with the `transform-origin` property. The default value is the middle of the element (a value of `50% 50%`); this is particularly useful for rotations.

Internet Explorer since version 10, Firefox since 16, and Opera since 12.1 have removed the need of prefixes for the `transform` property. All the other browsers at the time of this writing still need it, such as `-webkit-transform` for Safari on iOS.



3D transformations, such as `scale3d` or `matrix3d`, are not available in all browsers.

We can apply more than one transform at a time by declaring more than one function, space-separated. Don't declare more than one `transform` property, though, as the last one will override any previous ones. For example, to rotate an element 45 degrees while making it 10% bigger, we can use:

```
#element {
  -webkit-transform: rotate(45deg) scale(1.10);
  -moz-transform: rotate(45deg) scale(1.10);
  -o-transform: rotate(45deg) scale(1.10);
  -ms-transform: rotate(45deg) scale(1.10);
  transform: rotate(45deg) scale(1.10);
}
```



Remember that transformations don't affect other elements on the screen; therefore, if we translate or scale an element, other elements in the DOM will not be affected and will not be moved (unlike when we increase the margins).

Perspective

Setting a 3D perspective can be done using the `perspective()` transformation function in WebKit-based browsers only, or the special CSS property `perspective` and its prefixed versions, which take a value in pixels defining the distance from the viewer's perspective. If we use the latter option the perspective will be applied to the children of the element, but if we use the transformation function it will be applied only to the element itself.

Transform style

The transformation can act differently with regard to its nested (child) elements. We can control this behavior with the `transform-style` attribute, which has two possible values: `flat` and `preserve-3d`. If `flat` is used, the nested elements are flattened as if they were an image and the perspective is applied to that image. With `preserve-3d`, every nested element will have its own 3D perspective, as seen in Figure 11-17.

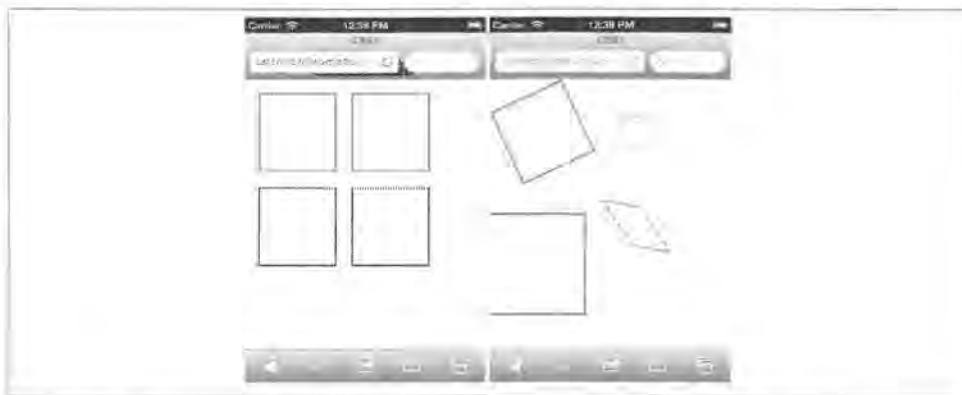


Figure 11-17. Sample of transformations applied to a couple of elements.

Backface visibility

Backface? What? An element in HTML has a backface? It can in CSS 3D transformations defining the `backface-visibility` CSS property. It is not what you might think (two faces in the same element), but the effect can easily be implemented.

The `backface-visibility` property can be defined as `hidden` or `visible`. If `hidden`, when we define a rotation of the *y*-axis of more than 180 degrees the element will disappear, and we can make another element showing a backface appear in its place.



All transformations can also be applied using JavaScript, by changing CSS styles or by using the `CSSMatrix` JavaScript class (prefixed on some browsers) and defining a couple of objects. The most simple and quick way is to define the transformation as a string and apply it to `element.style.transform` or its prefixed versions, such as `element.style.webkitTransform`.

The CardFlip pattern

This is one of the most “wow” visual features that Safari on iOS has brought to us. The CardFlip pattern allows us to show an element in a rectangular area and, when some event occurs, perform a transformation that flips the element as if it were a poker card and shows another element of the same size and in the same position as the backface.

Apple provides a full sample (<http://www.mobilexweb.com/go/cardflip>) that can be used as the base template for designing this kind of animation.

A simplified version (without all the prefixes) of the CardFlip sample looks like this:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Card Flip</title>
<style>
body {
    margin: 0px;
}

#container {
    height: 356px;
    width: 320px;
    background-color: rgba(56,108,179, 0.5);

    /* Disable tap highlighting on WebKit browsers */
    -webkit-tap-highlight-color: rgba(0,0,0,0);

    /* Give some depth to the card */
```

```

    perspective: 600;
}

.card {
    position: absolute;
    height: 300px;
    width: 200px;
    left: 60px;
    top: 28px;

    transform-style: preserve-3d;
    transition-property: -webkit-transform;
    transition-duration: 1.5s;
}

.card.flipped{
    transform: rotateY(180deg);
}

/* Styles the card and hides its "back side" when the card
   is flipped */
.face {
    position: absolute;
    height: 300px;
    width: 200px;
    border-radius: 10px;
    box-shadow: 0px 2px 6px rgba(0, 0, 0, 0.5);
    backface-visibility: hidden;
}

.face > p {
    margin-top: 36px;
    margin-bottom: 0;
    text-align: center;
    font-size: 92px;
}

.front {
    color: rgb(78,150,249);
    background-color: rgb(34,65,108);
}

.back {
    color: rgb(34,65,108);
    background-color: rgba(78,150,249,0.5);
    /* Ensure the "back side" is flipped already */
    transform: rotateY(180deg);
}
</style>
<script>
function flip(event) {
    var element = event.currentTarget;

```

```

    /* Toggle the setting of the className attribute */
    element.className = (element.className == 'card') ?
        'card flipped' : 'card';
}
</script>

</head>
<body>
    <div id="container">
        <div id="card" class="card" onclick="flip(event)">
            <div id="front" class="front face">
                <p>♠ ♡<br> ♣ ♥</p>
            </div>
            <div id="back" class="back face">
                <p>♦ ♠<br> ♥ ♣</p>
            </div>
        </div>
    </div>
</body>
</html>

```

Analyzing the code, we see two `div` elements inside a container called `card`. One `div` is the “front” face and the other the “back” face. Both faces are positioned in the exact same position (as absolute elements), and the back side starts with a *y*-axis rotation of 180 degrees. Both faces also define themselves as hidden when backfaced.

When the user clicks the card container (with either the front or back face displayed on the screen), via JavaScript we apply (or not) the `flipped` CSS class, which rotates both elements 180 degrees around the *y*-axis. And *voilà!* Only one face will be at the front at any given time; the other will be automatically hidden.



Remember that for the sake of simplicity, the previous example doesn't include all the necessary prefixed versions. To make it work on all browsers, you should clone every CSS transform style with a `-webkit-`, `-moz-`, and `-o-` prefix.

This process is done with a beautiful, smooth animation, which you can't quite see in Figure 11-18.

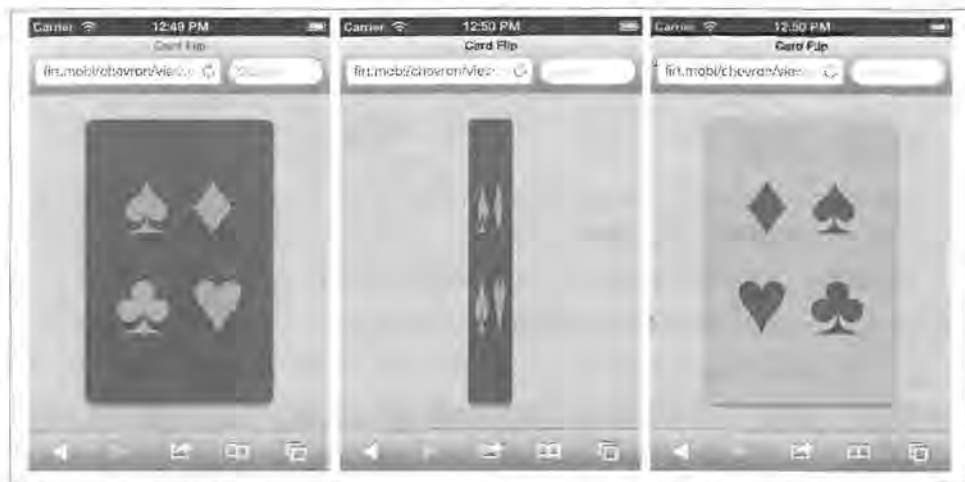


Figure 11-18. With 3D flipping you can use a beautiful 3D effect to display the backface of an element.

Transitions

A *transition* is just an automatic animation that takes place when a CSS property's value changes. The property must be defined by the browser as able to animate. There isn't an official list of properties that animations will work on, but the general policy is that any attribute with numerical or color values should be animated using transitions (we can test different attributes with transition ability using *animatable* (<http://leaverou.github.com/animatable>)). There are also a few exceptions, like *visibility* (a discrete property) and the *transform* property, which can animate any of the inner functions, such as *rotate()* or *translate()*. A version of the CSS transitions spec (<http://www.w3.org/TR/css3-transitions>) is available.



Remember, these transitions are defined entirely using CSS: we are not using JavaScript or any other technique to create the animations. This may sound a bit strange, but it is a simple and powerful technique. In some browsers, it's hardware accelerated and/or managed by an internal browser timer so it gets the best frame rate possible.

To create a transition, we should:

1. Define the transition properties (duration, delay, where to apply, timing function) in the element(s) we want to animate.
2. Change the values of the attributes of the element(s) to animate using JavaScript, or apply classes to, or remove them from the element.

3. Verify that the animation is working.

Sounds simple, right? Let's do it.

Animation properties

An animation can be defined using the shortcut property transition (and all prefixed versions) with the following syntax:

```
transition: property duration timing_function delay [, ...];
```

We can also use the specific properties listed in Table 11-2.

Table 11-2. CSS transition properties

Property	Description
transition-property	Defines which property or properties to animate. We can use a comma-separated list, or the constant value <code>all</code> .
transition-duration	Defines the duration of the transition. The value can be 0 (no animation) or a positive value in seconds (using <code>s</code> as the unit) or milliseconds (using <code>ms</code> as the unit). If we want to define different timings for each property, we can use a list of comma-separated values in the same order as the <code>transition-property</code> value.
transition-delay	Defines the offset delay of the animation, beginning from the time when the property was changed. This can be defined in seconds or milliseconds (using <code>s</code> or <code>ms</code>), and the default value is 0. If a negative value is used, the animation starts immediately but with some of the animation already done.
transition-timing-function	Defines the function used to calculate intermediate values from the initial to the finishing value of the property. You can use the CSS <code>cubic-bezier()</code> function, or any of the following constants: <code>ease</code> , <code>linear</code> , <code>ease-in</code> , <code>ease-out</code> , and <code>ease-in-out</code> (the most commonly used and default value.)

Internet Explorer from version 10, Opera from 12.1, and Firefox from 16 have removed the need of a prefixed version for the transition's properties.



Some modern browsers support staircase-timing functions that separate the animations in repetitive steps. The available functions are `steps()`, `step-start()`, and `step-end()`. These functions are in the standard draft but are not implemented in all browsers.

For example, the following code (with all the prefix versions) produces a fade-in, fade-out animation:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Fade Sample</title>
<style>
```

```

#box {
  width: 200px;
  height: 200px;
  background-color: red;
  -webkit-transition: opacity 2s;
  -moz-transition: opacity 2s;
  -o-transition: opacity 2s;
  transition: opacity 2s;
}

.hide {
  opacity: 0;
}

</style>
<script>
function fade() {
  var box = document.getElementById("box");
  box.className = (box.className=="hide") ? "" : "hide";
  box.innerHTML = box.className;
}
</script>

</head>
<body>

<h1>Fading</h1>
<input type="button" onclick="fade()" value="Hide-Show" />
<div id="box">
</div>
</body>
</html>

```

We can do similar transitions for resizing, relocation, color changes, or even 3D transitions using the transform properties that we saw earlier.

Transition ending

The transition ending can be listened for from JavaScript just like any other DOM event, using `addEventListener`. You can then initiate another transition or do something else when you are sure that the animation has finished. The event to listen for is called `transitionend`. The main problem is that in some browsers the event name is prefixed, so we really need to listen to different possible events and apply all of them to the same handler. The possible names are `transitionend`, `webkitTransitionEnd`, and `otransitionend`.

We can listen for this event using the following code:

```

box.addEventListener('webkitTransitionEnd', transitionEnd, false);
box.addEventListener('transitionend', transitionEnd, false);

```

```
box.addEventListener('otransitionend', transitionEnd, false);

function transitionEnd(event) {
    alert("Finished transition");
};
```

Animations

Transitions are great and are the simplest way to create animations for smartphones and tablets. If you need finer animation control at the keyframe level, you can use the CSS animations module (<http://www.w3.org/TR/css3-animations>). To be completely honest, I thought this was too much to be handled only by CSS—a nonprocedural and nonmarkup language—but it works great.



There are some visual tools available that will make creating CSS animations easier, such as Sencha Touch Animator (<http://www.sencha.com/products/animator>) and Adobe Edge Animate (<http://html.adobe.com/edge/animate>).

CSS animations are done with the shortcut property `animation` (and prefixed versions), which has the following syntax:

`animation: name duration timing_function delay iteration_count direction`

As you've probably guessed, there are also specific properties for each possible value, listed in Table 11-3.

Table 11-3. CSS animation properties

Property	Description
<code>animation-name</code>	Provides the name of the animation to be used by the keyframes.
<code>animation-duration</code>	Specifies the duration of the animation, in seconds or milliseconds.
<code>animation-timing-function</code>	Defines the function used to calculate intermediate values between the initial and final values of the property. You can use the CSS <code>cubic-bezier()</code> function or any of the following constants: <code>ease</code> , <code>linear</code> , <code>ease-in</code> , <code>ease-out</code> , and <code>ease-in-out</code> (the most commonly used and default value). You can design your own <code>cubic-bezier()</code> variant using Ceaser (http://matthewlein.com/ceaser/).
<code>animation-delay</code>	Defines the offset delay of the animation, beginning from the time when the property was changed. This can be defined in seconds or milliseconds, and the default value is 0. If a negative value is used, the animation starts immediately but with some of the animation already done.
<code>animation-iteration-count</code>	Defines how many times the animation will be repeated. This can be 1 (the default value), any integer value, the special constant <code>infinite</code> , or a float value.
<code>animation-direction</code>	Defines whether the animation will play in forward direction (<code>normal</code>) or in alternate mode, playing forward on even iterations and in reverse on odd iterations.

Property	Description
animation-fillmode	Defines whether the animation changes the element's styles before or after the animation. The default value none means no changes are done when the animation is not active. The forwards value means the last execution keyframe will stay, backwards will apply the initial keyframe before the animation starts, and both applies both behaviors.
animation-playstate	Defines if the animation is running or paused.

After reading this list of properties you are probably asking yourself a few questions. Where is the animation defined? What will be animating? For these, the CSS @keyframes at-rule comes into play.



If you are moving or scaling an object and you want it to be animated, it is better to use the performance-accelerated transform property rather than the usual CSS properties, such as top, left, or width.

@keyframes

To define how the animation will work and what it will do, we need to define a special CSS at-rule called @keyframes. This rule is followed by the animation name (the one specified in animation-name).

@keyframes and Prefixes

The big problem with the @keyframes at-rule is that it needs to be prefixed in some browsers—and that means we need to clone the entire keyframe declaration for WebKit-based browsers and for Firefox and Opera.

Opera 12.0 supported only the prefixed version, but the prefix is no longer required from version 12.1, so we can safely ignore it. Similarly, Firefox has removed the need of a prefix from version 16. So, as long as we are not targeting older Firefox editions, we can just define the WebKit prefix for iOS, Android, and other browsers and use the non-prefixed version for Internet Explorer 10 and newer versions of Firefox and Opera. Eventually WebKit browsers will also stop requiring the prefix, but to support 2011 and 2012 browsers we need at least:

```
@keyframes name {
  /* All declarations */
}
@-webkit-keyframes name {
  /* All declarations */
}
```

Inside the `@keyframes` at-rule, we need to specify as many selectors or animation groups as we want keyframes. The selector is defined by a percentage value or the constants `from` (equivalent to 0%) and `to` (equivalent to 100%). Inside each selector, we define all the properties and values that we want at that point in the animation. We can also define the timing to use in every animation group using `transition-timing-function`.



When the animation finishes, the original values are restored. The elements will not maintain the last keyframe values after the animation stops unless we define `animation-fillmode: forwards`.

For example, the following sample moves a `div` in a square path. For reasons of simplicity we are just using the standard code without prefixes, but remember that to support most browsers today we need at least the `-webkit-` prefixed version:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Animation Sample</title>
<style>

#box {
  width: 200px;
  height: 200px;
  background-color: red;
  position: absolute;
  top: 0px;
  left: 0px;
}

.squareAnimation {
  animation-name: squarePath;
  animation-duration: 4s;
  animation-timing-function: linear;
  animation-iteration-count: infinite;
}

@keyframes squarePath {
  /* We can use 0% or "from" as selector */
  from {
    top: 0px;
    left: 0px;
  }

  25% {
    top: 0px;
    left: 100px;
  }
}
```

```

50% {
    top: 100px;
    left: 100px;
}

75% {
    top: 100px;
    left: 0px;
}

/* We can use 100% or "to" as selector */
100% {
    top: 0px;
    left: 0px;
}
}

</style>
<script>
    function start() {
        // When we apply the -webkit-animation attributes, the
        // animation starts
        document.getElementById("box").className = "squareAnimation";
    }
</script>

</head>
<body onload="start()">

<h1>Moving over a square path</h1>
<div id="box">
</div>
</body>
</html>

```



If we define the animation attributes in the element from the beginning, the animation will begin when the page loads. The best solution is to define animations as classes and, when we want to start an animation, apply that class to the element.

So, to start the animation, we apply the class, and if we want to stop it before it reaches the ending value, we should assign an empty value to the `transform-name` property.

We can define one animation that changes several properties, or use different animations with different names at the same time, each changing a single property.



AliceJS (<http://blackberry.github.com/Alice/>) is an open source JavaScript library that can be used to easily create high-performance CSS3 transitions and animations, including twirl, page-flip, bounce, and dance effects.

Animation events

As with transitions, we can listen for the events `animationstart`, `animationiteration`, and `animationend`. When fired, they will send an `AnimationEvent` object as a parameter. There is no event to capture each keyframe change.

The event object has the special properties `animationName` and `elapsedTime`, whose value is given in seconds.

Remember that animation events, like transition events, need prefixed versions for some browsers.

CSS Filter Effects

The CSS filter effects specification is not part of any standard at the time of this writing, but filter effects do work on some browsers, such as Safari on iOS since 6.0, Chrome for Android since version 25, and the BlackBerry 10 browser. They use the `filter` property (`-webkit-filter` is the only prefixed variant available at the time of this writing), which accepts one or more filter functions that can be applied (space separated).



Internet Explorer used to support a nonstandard `filter` property, but it's not compatible with this CSS filter specification. From version 10, Microsoft doesn't recommend the usage of the `filter` property.

Filters come from SVG, and the list of filters already available for usage includes `sepia`, `grayscale`, and `blur`. In compatible browsers, we can even animate or transition within filters.

If you want to try all the possible filters dynamically in a compatible browser, you can point to mobilexweb.com/go/filters.

For example, we can define:

```
-webkit-filter: blur(5px) grayscale(.5) opacity(0.66)  
hue-rotate(100deg);
```

CSS Regions and Exclusions

CSS regions (<http://dev.w3.org/csswg/css3-regions>) and CSS exclusions (<http://www.w3.org/TR/css3-exclusions>) and shapes are two modules proposed by Adobe to cover some design issues with HTML.

With regions, we can define arbitrary regions—not necessarily contiguous—through which the content can flow. Therefore, a long text can flow between different areas on the screen, as you can see in Figure 11-19. With exclusions, we can define “islands” around which the text should flow for richer layouts, as seen in Figure 11-20.

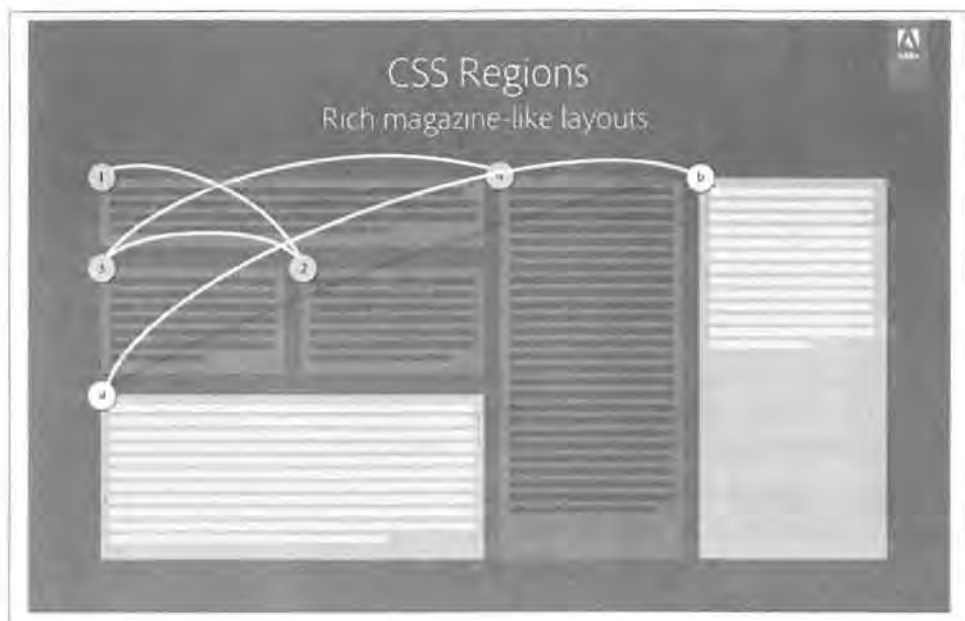


Figure 11-19. With CSS regions we can define regions through which the content will flow, such as in a printed magazine layout.

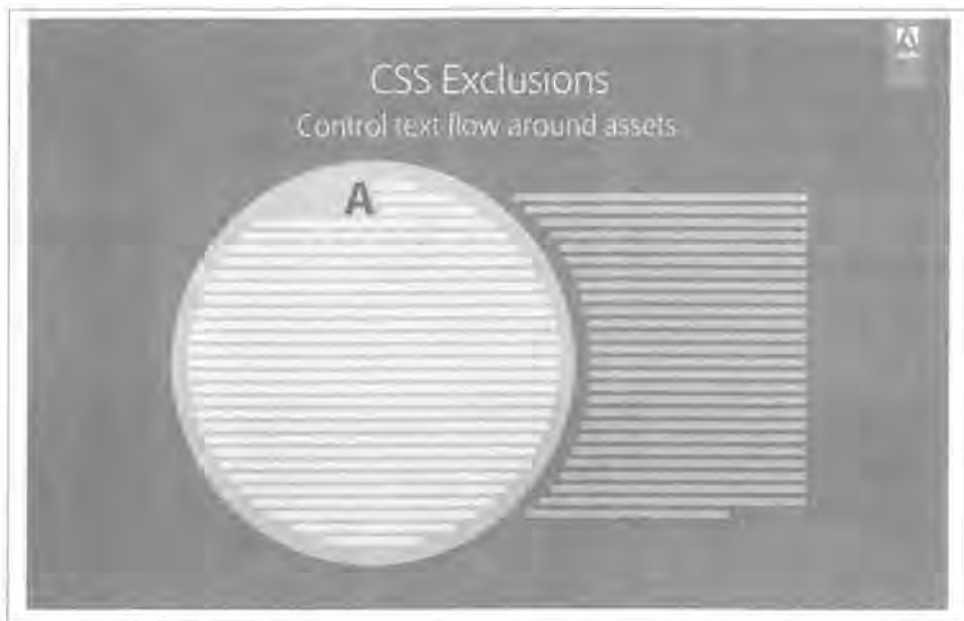


Figure 11-20. CSS exclusions and shapes allow us to define islands for text to flow around for rich layouts, as we can see in this example from Adobe.

At the time of this writing, the only browser supporting these specifications is Internet Explorer 10, but we can expect better support in the near future. For updated compatibility information, see Mobile HTML5 (<http://mobilehtml5.org>).

Adobe's commitment to bring us all a richer Web by proposing new standards can be viewed at Adobe's website (<http://html.adobe.com/webstandards>).

New CSS Values and Units

The CSS3 values and units module (<http://www.w3.org/TR/css3-values>) includes new units and methods that we can use in most CSS declarations.

As we saw earlier in this chapter, we now have angle units, such as `deg` for degrees, and many other new features.

Safari on iOS 6 became one of the first mobile browsers to support mathematical expressions through the `calc()` function. We can use basic math expressions, such as additions or divisions using different units. For example, we can define something as `width: calc(100% - 24px)`. Internet Explorer, Chrome from version 25, Firefox, and BlackBerry 10 also support the `calc()` function.

Usually, browsers supporting these new values also accept new viewport units that are attached to the viewport, including `vw` (equal to 1% of the width of the initial containing block), `vh` (equal to 1% of the height), `vm` (equal to the smaller of `vw` and `vh`), and `vmax` (equal to the larger of the two).

While these new values and units may be useful, you need to make an analysis of current compatibility before deciding on their usage.

CHAPTER 12

JavaScript Mobile

Designing for the Web is about more than content and presentation—users expect websites to be interactive, responding to their choices. Fortunately, although it has some limitations, the JavaScript we use in developing for the mobile web is similar to that used in desktop web development.

We have already talked a bit about the WAP 1.1 standard scripting language, *WMLScript*. Don't worry, we won't go any deeper into this obsolete language.

Thankfully, a couple of years after this standard was released, mobile browsers started to add some support for JavaScript (or ECMAScript, to be totally correct). The standards for mobile scripting are more difficult to define than the standards for CSS. The great benefit is that, excepting some bugs, JavaScript can check at runtime whether some feature, object, or API is available, so we can code for different “standards.”



The standard name for JavaScript is ECMAScript because it is defined by the ECMA (an international, private, nonprofit standards organization). There are three well-known dialects on the market: JavaScript (trademark of Oracle, licensed now to the Mozilla Foundation), ActionScript (trademark of Adobe), and JScript (trademark of Microsoft). At the base, they are the same language, and everyone adds new behavior. In practice, the name JavaScript is so common in the industry today that everyone—including Microsoft—uses it as the language name.

The only mobile-specific standard is called *ECMAScript Mobile Profile* (ESMP). It was defined by the Open Mobile Alliance (OMA), like XHTML MP and WAP CSS. In my 10 years in the mobile web world, I have never heard a developer or a company talking about ESMP. It is a subset of the ECMAScript language and is really just JavaScript with some features left out and some other features added in (imported from WMLScript).

The XHTML MP 1.2 OMA standard has recommended using ESMP, but we will just use standard JavaScript code, as clean as possible, and it will work everywhere.



VBScript is a similar language (created by Microsoft, based on Visual Basic), but it is compatible only with Internet Explorer for desktop. VBScript is not compatible with any version of Internet Explorer for mobile devices.

There are many versions of JavaScript available (at the time of this writing, from 1.0 to 1.8). The most stable version for all browsers (from low-end devices to social phones) is 1.5, and this is the version we should care most about. Newer versions only work in the latest editions of some browsers, and the additions aren't worth the incompatibility. JavaScript developers often aren't aware of different languages' versions, so we will talk about feature compatibility here instead of comparing version numbers.

Some modern browsers support ECMAScript 5, including the strict mode, such as Safari on iOS since 5.0, the Android browser since 3.0, Chrome, and the BlackBerry browser since 7.0.



A number of other scripting languages have appeared in the last years to replace or enhance JavaScript, such as Dart, CoffeeScript, and TypeScript. While no mobile browsers currently support them, you can find compilers that will cross-compile these new languages to standard JavaScript code.

Coding for Mobile Browsers

When we talk about mobile JavaScript, we are talking about the same code you already know: a script tag including some code or a script tag with an external source:

```
<script>
// Code goes here
</script>

<script src="mysource.js"></script>
```



ECMAScript Mobile Profile defines two new types for the script tag, `application/ecmascript` and `text/ecmascript`, as the preferred types to use for ESMP-compatible code. Beyond that, `text/javascript` is the recommendation for XHTML MP documents and is the most compatible type to use for non-MP browsers. My recommendation is to carry on using the well-known `text/javascript`.

HTML5 Script Extensions

HTML5 includes new attributes to the `script` element that may improve performance on compatible devices. The `script` elements used to block the parallel download of any other resource, and that harms performance. While there are hacks and tricks to load a script asynchronously, HTML5 includes the `async` Boolean attribute that defines that the execution should be done asynchronously without blocking other downloads.



The `async` and `defer` attributes work in Safari on iOS since 5.0, the Android browser since 3.0, the BlackBerry browser since 7.0, and Internet Explorer since 10, as well as Firefox and Google Chrome. IE9 supports only `defer`.

If we want to defer the execution of the script until the whole document has been parsed, we can use the `defer` Boolean attribute. Both `defer` and `async` work only if we are loading external JavaScript files using `src`.

Code Execution

You can execute JavaScript code in four different ways:

- From a `script` tag
- From an event handler
- From a link, using the *javascript:* URL protocol
- From a bookmarklet, using the *javascript:* URL protocol

Bookmarklets

A *bookmarklet* is a bookmark in the browser containing some JavaScript code using the *javascript:* URL protocol. When the user activates the bookmark, the JavaScript code is executed over the current document. This allows us to execute a wide range of testing, debugging, and other features over any web page.

There are bookmarklets on the Web that are large applications, encoded in a single line of JavaScript. One of my favorites for desktop usage is Readability (<http://bit.ly/WpPQIe>).

The main problem with bookmarklets in mobile devices is how to add them. In the desktop web, the main way is to drag a link with the JavaScript code to the bookmarks area. This cannot be done in a mobile device, though, so bookmarklets are only useful if you can manage or synchronize them from a desktop (via iTunes for iPhone).

There are a lot of bookmarklets for iPhone on the Web, including some that will show the source code of the page inside the mobile browser. These are only really useful for testing or debugging purposes, though, or for “only for geeks” features. Mobile Perf is a bookmarklet created by Steve Souders that will help us with tools to measure and work with performance; you can install it on your browser by following the instructions at Steve Souders’ website (<http://stevesouders.com/mobileperf/>).

Cloud-Based Browsers

We need to pay special attention to cloud-based browsers, like Opera Mini and the Nokia Xpress browser. Remember that these browsers render our websites on their servers and send compressed and compiled content to the clients. Some of these clients aren’t really browsers capable of rendering an HTML file or JavaScript code on the device itself. Therefore, all the code that executes before and during the loading process is executed on the server before it sends the rendered result to the browser. Any code executed later, such as because of a `click` event, will send a request to the server, which will instantiate the web state, execute the code, and send back the changes.



Some cloud-based browsers execute every line of JavaScript server-side; therefore, we should keep dynamic updates to the minimum, as all of this code will require a server postback.

Some other cloud-based browsers, such as Amazon Silk, execute our JavaScript partially on the server side and partially on the client side, using a nonpublic algorithm that is advertised as making smart decisions to increase the browsing speed.



For older mobile browsers that do not support JavaScript, or browsers with JavaScript disabled, we can use the `noscript` tag. Only those browsers will display its content.

Nokia Mobile Web Library

As we’ll see in Chapter 16, when we are working with the Nokia Xpress browser and developing cloud-based Nokia S40 web apps, we have a limited series of functions that we can call that will be executed on the client side, without a server postback. We can make these calls using the Nokia Mobile Web Library (MWL), a JavaScript framework that exposes functions in the `mwl` global object.

If we call `mwl` functions directly inside events, these functions will be executed on the client side. Any other calls will be sent to the server. Available functions include `show()` and `hide()` for changing an element’s display properties, `addClass()` and `removeClass()`, and `timer()` to initiate a time-based operation. For example:

```
<input type="button" onclick="mwL.hide('#content1');  
mwL.show('#content2');" value="Show content 2">
```

For a complete reference of MWL, check out <http://mobilexweb.com/go/mwL>.

JavaScript Debugging and Profiling

Debugging and profiling JavaScript for mobile browsers and mobile web apps can be a little more complicated than we are used to. Doing the process on the device itself usually is not a good idea, as on-device tools are just not good enough.

In Chapter 18, we are going to look at all the tools available for remote debugging and profiling sessions on many platforms, as well as other hacks for the purpose.

Battery Consumption

You may be used to working with milliseconds and/or bytes in terms of performance and measures. Do you believe me if I say that now we need to add the unit *joules* to our calculations? Joules is a measure of energy consumption—it reminds us that our JavaScript code will consume battery power. And we know that a low battery is one of the biggest problems for every mobile user.

A team from Stanford University, working in conjunction with Deutsche Telekom R&D Laboratories, has published a study on how much battery power mobile web browsing consumes; it's called "Who Killed My Battery: Analyzing Mobile Browser Energy Consumption" (<http://mobilexweb.com/go/battery>).

While the abstract has lots of useful information, I want to summarize a few key points here:

- Unused, orphan JavaScript code can consume battery power for nothing.
- Not providing a mobile-specific website and viewport means more power consumption, as the browser needs to calculate resizing actions and download big resources.
- Parsing a big JavaScript framework, such as jQuery, may consume around 0.01% of a fully charged battery on some devices.

So, keep your JavaScript code to the minimum—and that includes using frameworks (such as jQuery) only when you really need to, and not just because you're used to it.

Background Execution

When we see a mobile browser on a smartphone or a tablet, our first thought is that it works in a similar way to a desktop browser. However, browser behavior can be really different, and these differences impact our code's lifetime.

Because of concerns about battery usage and general UI performance, most mobile operating systems, even on tablets, freeze all applications running in the background. Another important difference is that usually closing and minimizing are the same operation. Therefore, from a user's perspective, if she goes to a different app or to the Home screen, the browser (and all the open websites) will be frozen. That means no JavaScript code will be executed in the background. While some exceptions apply, this is the expected behavior in most mobile browsers.

A similar situation applies when the phone is in sleep mode, or when the browser is still in the foreground but the user has switched to a different tab or window. Usually, the tabs or windows that are not in the foreground are not being executed and may not be in memory at all.

Safari on iOS, Firefox, the Android browser, and some other browsers freeze all JavaScript code when the user changes tabs/windows, locks the phone, or minimizes the browser. Opera freezes our code when the user locks the phone or minimizes the browser, but it maintains execution if the user switches to a different tab.

On the other hand, Google Chrome for Android keeps our website alive while the browser is still in memory, but if we are using timers the frequency slows down to one second.



On most platforms, we can only be sure that our JavaScript code will be alive when the browser is in the foreground and our website is in the current tab or window. Typical desktop behavior that uses Ajax calls or similar techniques to update content frequently while in the background will not work in most phone and tablet browsers.

The possible situations that can change a page's visibility status and affect the execution of our code are:

- The browser is in the foreground and our website is in the active window or tab. In this situation, our JavaScript is being executed.
- The browser is in the foreground and our website is not in the active window or tab. However, it is still in memory, usually in a frozen state (no code is being executed). The code will continue its execution from the point at which it was frozen if the user reactivates the window or tab.

- The browser is in the foreground, but our website is not in the active window or tab and is not in memory. This may be because the browser needed memory for other websites, or because the window or tab was opened in a previous browser session. The JavaScript context will be lost in this situation, and when the user reactivates the window or tab our website will be reloaded and will start from scratch.
- The browser is in the foreground, but the device is asleep.
- The browser is in the background and our website is still in memory, usually frozen without any execution possibilities. If the user goes back or opens the browser again, the session will be restored.
- The browser is in the app history (so to the user it appears to still be open), but it has been removed from memory by the operating system, which needed the memory for other purposes. If the user goes back or opens the browser again, it will start from scratch and will reload previous tabs or windows and their contexts.
- The browser is in the background, and it's compatible with background JavaScript execution; in this situation, our code is still being executed.



At the time of this writing, the only mobile operating system that supports having two apps in the foreground at the same time is Windows 8 for tablets, when using the snap mode (described in Chapter 11). Android devices can have background or service apps, but browsers don't usually fall into this category.

Status Detection

On most devices, timers (and all JavaScript execution) are paused when the web page is sent to the background. I have an iPod touch, and in Safari I always have my email open in one of the eight possible tabs (or windows). When I want to browse to another website, I change to another tab but leave that one open. That means my email can be frozen for several hours or even days, until I go back to that tab. As developers, this raises an important issue: when our web pages are put into the background, how can we detect when they should “wake up” again?

The Page Visibility API

The only standard way to capture events indicating when our website is being activated or deactivated is the Page Visibility API (<http://www.w3.org/TR/page-visibility>). With this API we can check the `document.hidden` property to see whether the document is actually hidden (`true`) or visible (`false`). It also exposes the new event `visibilitychange` that we can use to detect changes in visibility.



The Page Visibility API also exposes a `document.visibilityState` property that will give us a value of `hidden`, `visible`, `prerender`, or `unloaded`.

The Page Visibility API is not available in every browser. At the time of this writing, it is prefixed in some browsers, such as Google Chrome, Firefox (before 18), and Internet Explorer 10. In these browsers, the `document.hidden` property can be accessed through `document.webkitHidden`, `document.mozHidden`, or `document.msHidden`, and the event name is `webkitvisibilitychange`, `mozvisibilitychange`, or `msvisibilitychange`. Safari on iOS (6.0) doesn't support this API at the time of this writing. Currently, only Opera Mobile since 12.1 and Firefox since 18 support the standard, unprefixed version.



Don't confuse the Page Visibility API with the `pageshow` and `pagehide` events available in some browsers on the `window` object. These events have nothing to do with the visibility of our website; they are equivalent to `load` and `unload` but with more control when the page is retrieved from the local cache.

When we detect that our website is not active anymore, we can:

- Store state to be restored on the next load, if the browser or website is deleted from memory.
- Stop timers and animations.
- Pause a game.
- Release big resources that could potentially get our website kicked out of memory.
- Stop server calls.
- Log out the user from the server.
- Close sockets or other server connections.

Remember that when we detect that the website is not on the screen anymore, we should not update the UI, such as showing a dialog. We can send Ajax requests to the server without any problem, though, to inform it that the user is leaving the page for now.

The following code snippet illustrates use of the Page Visibility API:

```
var eventName = "";

if (document.webkitHidden!=undefined) {
    // webkit prefix detected
    eventName = "webkitvisibilitychange";
} else if (document.mozHidden!=undefined) {
    // moz prefix detected
```

```

    eventName = "mozvisibilitychange";
} else if (document.msHidden!=undefined) {
    // ms prefix detected
    eventName = "msvisibilitychange";
} else if (document.hidden!=undefined) {
    // standard prefix detected
    eventName = "visibilitychange";
} else {
    // API not available
}

function visibilityChanged() {
    if (document.hidden || document.mozHidden || document.msHidden ||
        document.webkitHidden) {
        // Our website has just hidden
    } else {
        // Our website is back in the foreground
        // In some situations, the website will load from scratch
    }
}
document.addEventListener(eventName, visibilityChanged, false);

```

Wakeup detection using timers

If the Page Visibility API is not available, such as in Safari on iOS up to version 6.0, we cannot detect when the user is hiding our website. However, we can detect when the user goes back to the website.

Neil Thomas, a software engineer from Google working in the Gmail for Mobile team, has published a very simple and clever solution using a high-frequency timer and a global variable for calculating the time elapsed between calls to that timer. Because the timer will not fire when the application is in the background, if we detect that the delta time from the last execution is greater than a certain threshold value we can assume that the timer firing again indicates that the application has just woken up from hibernation.



Remember to use a large value for the threshold after deciding that the page has gone to sleep. Otherwise, depending on the tasks being done, the engine behind the browser, and the device hardware, it may take longer for the JavaScript code that's executing to complete than the time defined for the timer.

This is Thomas's public code (with a little variation from me). An explanation can be found at www.mobilexweb.com/go/timers:

```

// The time, in ms, that must be "missed" before we
// assume the app has been put to sleep.
var THRESHOLD = 10000;

```

```

var lastTick_;
var detectWakeFromSleep_ = function() {
    var now = new Date().getTime();
    var delta = now - this.lastTick_;
    if (delta > THRESHOLD) {
        // The app probably just woke up after being asleep.
        notifyWakeFromSleep(delta/1000);
    }
    lastTick_ = now;
};

```

In the `notifyWakeFromSleep()` method, you can decide what to do based on the received parameter telling you how many seconds have passed since the last active state. You may want to do different things if the delta time is 10 seconds or 1 day (86,400 seconds). For example, after a big delta you might want to show a warning or a loading animation while new results are fetched using Ajax.



There is one situation where we won't have the opportunity to wake up. If the device is running out of memory and our page is in the background, it is possible that the browser will delete the page state to release memory, and when the user comes back to it, our page will be loaded by URL as a new session.

Remember that after waking from sleep, the document and the script are in the same state (including their HTML content and JavaScript variables) as they were before going to sleep, unless the operating system or the browser has decided to remove it completely from memory, in which case the website will load from scratch.

Background Tab Notification Trick

While it can be an annoying behavior from a user's perspective when the page is active, an old HTML mechanism allows us to define a meta tag to reload a window automatically every *n* seconds. Some browsers, such as Safari on iOS, allow us to use this hack to automatically reload inactive tabs and keep them updated:

```

<!-- Updates the page every 1 minute -->
<meta http-equiv="refresh" content="60">

```



In Chapter 15, we will cover the Web Notifications API, which allows us to send a background notification to the user on compatible devices.

On iPad devices, we can use the `title` element to update the UI and capture the user's attention. This behavior is useful on tablets, as they usually show a tab UI, while on

smartphones the titles of any windows that are not in the foreground are hidden from the user's view.

With this technique, the page will reload on the iPad and the inline scripts and the onload event will be executed in the background tab. We can change `document.title` to send information to the user via the tab's title. No other event or timer will be executed after the onload event until the user goes back to activate the tab.

The problem is how to remove the behavior when the page is in the active tab, so the user doesn't get the reload effect while our website is in the foreground. Every time we set the content attribute dynamically in the meta tag, the browser starts counting again.

To make the trick work on the iPad, we can start a chronometer that will shift the refresh meta tag *n* seconds on every execution. While the page is still active, our chronometer will be executed and the reload action will be shifted every *n* seconds. When the page goes onto a background tab, the chronometer will not be fired and the refresh meta tag will trigger, refreshing the page:

```
<!-- Updates the page every 1 minute -->
<meta http-equiv="refresh" content="60" id="metarefresh">

<script>
// iPad background tab notification trick
var mr = document.getElementById("refresh");
setInterval(function() {
    mr.content=mr.content; // Shift the reload operation
}, parseInt(mr.content)/2); // Every 30 seconds in our example
</script>
```

Background Execution Compatibility

Table 12-1 shows how mobile browsers react to background JavaScript code in different scenarios.

Table 12-1. Background execution behavior in non-cloud-based browsers

Browser/platform	When device is locked	When tab/window is not active	When browser is in background	Supports Page Visibility API
Safari on iOS	Paused	Paused	Paused	No up to 6.0
Android browser	Paused	Paused	Paused	Yes from 4.0
Chrome for Android	Executing	Executing	Executing	Yes
Nokia Browser for Symbian	Paused	Paused	Paused	No
Nokia Browser for MeeGo	Paused	Paused	Paused	No
BlackBerry browser for smartphones	Paused	Executing	Paused	No

Browser/platform	When device is locked	When tab/window is not active	When browser is in background	Supports Page Visibility API
BlackBerry browser for PlayBook and BB10	Paused	Executing	Paused	Yes
Internet Explorer	Paused	Paused	Paused	Yes from 10
Opera Mobile	Paused	Executing	Paused	Yes
Firefox	Paused	Paused	Paused	Yes

Push Notifications

Most mobile platforms—including iOS, Android, BlackBerry, and Windows Phone—support push notification services for native installed apps. This kind of service is usually the way to alert the user of some update while the app is in the background or closed.



On iOS the notification can include a badge number, a sound reproduction, and/or a message alert to display; on Android, it includes a message; on BlackBerry it can include an icon change; and on Windows and Windows Phone it can include a message, a badge number, and a live tile update.

Websites can't use push notification services at the time of this writing, but we can definitely use this technique when creating native web apps, such as Apache Cordova/PhoneGap apps. This behavior is outside the scope of HTML5 and needs to be created in native code, per platform. The technique usually involves the following steps:

1. Get the user's authorization to be notified using native code.
2. Receive a hash authorization string.
3. Send the hash to our own web server and store it mapped to the current user.
4. When we need to send a notification, we create an HTTP/S request to a push notification server with the hash string and the notification details.
5. The notification server sends the notification to the device directly (or the device pulls notifications frequently from the server).



There are some multiplatform solutions that will help us to send notifications to all our users with one simple API, such as Urban Airship (<http://urbanairship.com>).

Supported Technologies

We are going to test JavaScript compatibility in the following pages, but you should be aware that making JavaScript work requires more than just support for the language. There are many technologies (or APIs) that are bundled with JavaScript, but they are optional and will not work on all devices.

The Document Object Model

The DOM is a set of conventions for manipulating, browsing, and editing XML and HTML documents using a set of API conventions that may be implemented in many languages. In fact, although many developers think that the DOM is a JavaScript thing, this is wrong. There are DOM APIs for PHP, .NET, Java, and many other languages.

Even if you've never heard about the DOM, odds are good that you've used it. If you've used the well-known `document.getElementById()` function, for example, you were using the DOM.

Today, the DOM is a W3C specification with several versions available. The most compatible versions for web use are the Level 3 specifications.

With the DOM, we can browse the HTML document structure and make changes and additions dynamically from JavaScript without refreshing the page.

A basic mobile browser can be JavaScript-compatible but without DOM functionality. There are also some browsers that allow us to browse the document tree but not to modify it on the fly, such as older BlackBerry browsers.

Devices shipping from 2010 usually don't have any problem with DOM management.

The Selectors API

The Selectors API (<http://www.w3.org/TR/selectors-api/>) is an extension to the DOM that allows us to use CSS selectors to retrieve an element result list from the DOM, à la jQuery. This mechanism is very popular when using the jQuery JavaScript library, and the API is included natively as part of most modern mobile browsers.

A query is made using `document.querySelector(selector)` for unique results, or `document.querySelectorAll(selector)` for many possible return values. For example:

```
var items = document.querySelectorAll("ul.menu > li");
var option =
    document.querySelector("#form1 input[type="radio"]:checked");
```

JSON

JavaScript Object Notation (best known as JSON) is a lightweight data interchange format known to be compatible with almost every language in common use. It is sometimes used in JavaScript as a replacement for other transport formats, like XML.

JSON can be used in Ajax requests or to store and load information on the client side. ECMAScript 5 includes a native JSON object that is also included in other older browsers. This object allows us to convert standard objects, arrays, and JavaScript variables to a string JSON format using `JSON.stringify()` and to convert JSON string files to objects again using `JSON.parse()`.

Most modern smartphone and tablet browsers support this object directly, and for older devices we can use a JavaScript polyfill (<https://github.com/douglascrockford/JSON-js>).

Binary Data

Because of the need for binary data in WebGL—the 3D canvas—modern browsers include a way to manage binary data efficiently, known as *typed arrays*. Even non-WebGL browsers, such as Safari on iOS, support these type extensions. The list of possible types includes `DataView`, `ArrayBuffer`, `Float32Array`, `Int32Array`, and `Uint8Array`. Typed arrays work pretty much the same as normal JavaScript arrays, but their execution is much faster.

We can use these new data types with the Canvas, WebGL, XHR2, Workers, Sockets, and File API, as we will see later in this book.

Web Workers

Web Workers (<http://www.w3.org/TR/workers>) is a W3C specification that allows JavaScript to create working threads instead of executing all the code in the main UI thread, shared with the browser's rendering engine.

The specification defines two kinds of workers: *workers* and *shared workers*. Creating a worker allows a script to create an isolated thread that can communicate bidirectionally with the opener script and it has its own isolated context. A shared worker can be accessed by different scripts in the same domain that are working in different contexts, such as different tabs, windows, or iframes. Using the idea of ports, the shared worker can communicate with different executing scripts at the same time.



Shared workers have no real usage in mobile browsers because as we have seen, usually only one window or tab is active and running. Therefore, the need for shared workers is low, although they may sometimes be useful if we are using scripts in iframes.

Mobile browsers vary in terms of worker compatibility, and not every mobile browser supports shared workers.



We can verify current Web Workers compatibility with mobile browsers at mobilehtml5.org; with Modernizr, we can also query on the standard `webworkers` property and the `sharedworkers` and `blobworkers` extension properties.

Workers are not available in the Android browser up to 4.1, Internet Explorer up to 9, and Opera up to 10. Support was incorporated in Safari on iOS from 4.0, Internet Explorer from 10, Opera from 11, and the BlackBerry browser from 7, and workers are available in Google Chrome, the BB10 browser, and Firefox as well.

A worker is always an external JavaScript file that will be executed in an isolated thread, and that means the worker can't access any DOM object, such as `document`, or UI feature, such as `alert`. A worker is especially important for performance purposes to avoid blocking the UI on long calculations and operations, such as a long parsing process. While a worker can't access the DOM, it can send and receive information to/from the main thread through a Messaging API.

To create a worker, we just instantiate it:

```
var worker = new Worker('workercode.js');
```

To enable bidirectional communication, we use the `message` event and the `postMessage()` function:

```
worker.onmessage = function(event) {  
    // event.data has the information sent by the worker  
}  
worker.postMessage(dataToWorker);
```

The message can be a basic data type such as a string or a number, an array, or an object containing other serializable data types. We can't pass native objects, such as `document` or `window`, through the communication channel. Objects being transferred are copied and not shared, so your object will be duplicated on the other side.



Using an object with JSON syntax, we can create our own communication protocol between the main thread and the worker using different properties to define what we need.

Some browsers support a nonstandard feature that allows us to send and receive shared binary data using a prefixed alternate version of `postMessage()`, such as `webkitPostMessage()` on Google Chrome.

The worker is an external file that has a special `self` object referring to the worker itself. It uses the same Messaging API, including `onmessage` and `postMessage()`, to communicate with the main thread:

```
// workercode.js
self.onmessage = function(event) {
    // event.data has the information sent by the main thread
}
self.postMessage(dataToMainThread);
```

With a hack, some browsers allow the execution of a worker using a data URI or a blob element (binary data of a script) that can be created dynamically from a string file. Therefore, we can avoid the usage of an external file; however, this hack is not safe for mobile browsers today.



A worker can load external files using the `importScripts()` function and it can be terminated at any time using the `terminate()` method.

HTML5 APIs

With HTML5, JavaScript supports some new APIs for client scripting and document work. Mobile browsers are already adopting some of these new APIs, even though all the standards are still in discussion. We have already covered some of these APIs, such as Canvas, and in the next chapter we will focus on many others.

Native Web App APIs

There are other extensions available for native web applications on some devices, and many other JavaScript APIs are supported in installed applications.

These JavaScript APIs can include support for:

- Messaging
- Address book management
- Gallery
- Camera
- Calendar
- Device status information
- Native menus

Standard JavaScript Behavior

As you move into JavaScript on the mobile web, you'll want to test compatibility and use some old-fashioned features.



Even on JavaScript-compatible devices, a script might not work because of the user's (or company's) profile. For example, older BlackBerry devices have the option to disallow JavaScript from the browser or from the company policies. You should always present a non-JavaScript version of your site's functionality.

Standard Dialogs

JavaScript supports a list of standard dialogs that are undervalued in modern desktop websites, often being replaced by Dynamic HTML or UI libraries. They make great standard dialogs for use in mobile websites, though, as shown in Figure 12-1.

The list of available dialogs is:

- `alert`, for showing a message
- `confirm`, for receiving a Boolean response from the user
- `prompt`, for receiving a string from the user
- `print`, for sending the web page to the printer
- `find`, for invoking the find feature of the browser

The `find` dialog isn't really part of the standard, but it works in almost every non-IE desktop web browser on the market. This dialog receives three optional parameters: the text to find, a case-sensitive Boolean, and a backward Boolean. In general, it should be avoided in mobile browsers. Most of them don't even have a search feature.

The `print` dialog works only in recent versions of Safari on iOS supporting wireless printing, but the `alert`, `confirm`, and `prompt` dialogs are compatible with almost every JavaScript-enabled mobile device. I encourage you to use them when needed. Using a standard dialog will always be quicker, simpler, nicer, and more compatible than using any other solution for the same task.



The `prompt` standard dialog is not compatible with Internet Explorer 9 and 10 for Windows Phone, or with IE10 in Windows 8 Metro mode.

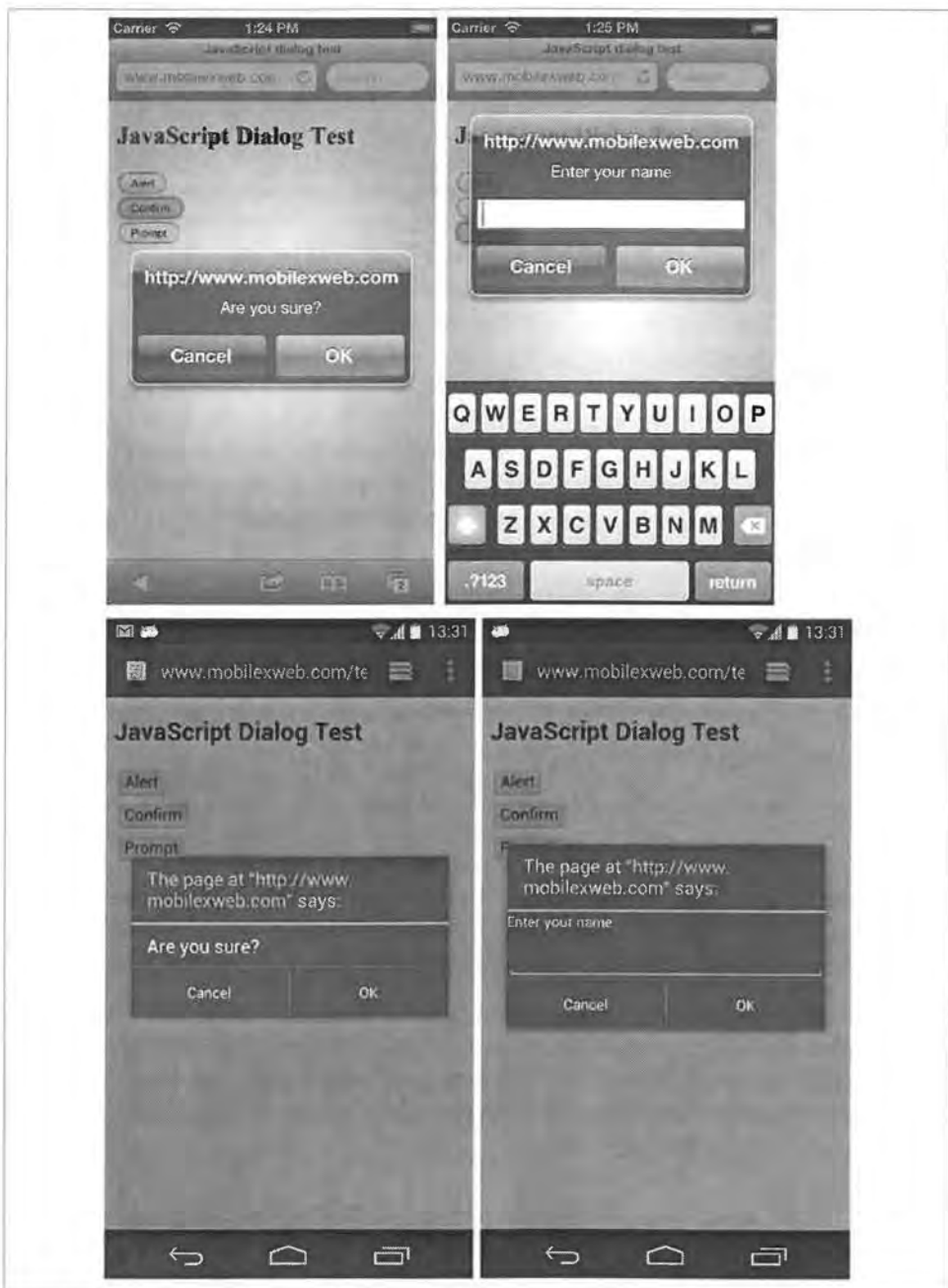


Figure 12-1. With standard JavaScript dialogs you get rich, multiplatform interfaces for free, using UI controls from the operating system.

Common problems with the alert dialog (and the others) include the usage of the carriage return for multiline text, and how to display text that's too big to fit in the available space. For the first problem, it is common in desktop JavaScript to use the `\n` (newline) escaped special character. (There are other special escape characters, too, like `\t` for tabulation.) Let's see what happens with both problems in mobile browsers. Figure 12-2 shows the use of a scrolling area that supports long text.



Figure 12-2. In some browsers, big alerts have scroll bars (or can be scrolled with a finger, on touch devices).



For usability reasons, if a device has a numeric keyboard, it is best to use a normal text input with numeric capabilities, rather than a selection list. Remember that a numeric keyboard is useful for numeric entries.

History and URL Management

JavaScript has a few standard mechanisms for browser history management: the `location` and `history` objects. The `location` object has several properties regarding the address, like `href` for the whole URL and `hash` for the anchor part of the URL, if present (the `#` and everything to the right of it). Changing the `location.href` property will redirect the browser to another page, on compatible devices. It has two useful methods: `reload()`, which refreshes the same page, and `replace(url)`, which sends the user to another page without creating a new history entry.

The `history` object has a few not-very-useful properties and three methods: `back()`, `go(number)`, and `forward()`. The `back()` method is the most commonly used, for emulating a back button:

```
<!-- As a button -->
<input type="button" onclick="history.back()" value="Back" />
<!-- As a link -->
<a href="javascript:history.back()">Back</a>
```

Remember that we are designing for mobile browsers, and sometimes the users will be browsing in full-screen mode without any browser buttons in sight. A link or button for going back will be more useful here than in desktop websites.

History API extensions

HTML5 has added new abilities to the `history` object that let us replace an existing entry or add a new entry in the history without affecting the page load process. While not a good idea, we can even create nonexistent URLs with this technique (it's not a good idea because if the user bookmarks or reloads the page, the URL will be invalid).

The new methods are `history.pushState(data, title, url)` and `history.replaceState(data, title, url)`. `data` is any object we want to store within the current state, and current data can be accessed through `history.state`. In these browsers, once we have pushed a fake URL into the history, we can detect when the user goes back using the `popstate` event of the `window` object.



`pushState()` does not use the hash (`#` sign) to change the URL; it will just create a fake complete URL in the history and in the URL bar.

The hashchange event

HTML5 also adds the hashchange event that we can bind to the window object: this event is triggered every time the hash part of the URL (after the # sign) is changed. So, we can use `location.href` to add a hash keyword after the current URL and then detect when the user goes forward and backward using hashchange.

Manipulating Windows

One of the most popular (and annoying) features of JavaScript in its early days was the usage of `window.open` to open the classic pop-up windows. For mobile browsers, the usage of this technique is not ideal, for many reasons. Many browsers can't open multiple windows (although Figure 12-3 shows one that can in action), and we cannot define any attributes for the pop-ups; they will just be full-sized, like the main window. Communication between the opener and the pop-up also often does not work well. Finally, closing pop-ups can be problematic on browsers that treat the new window as a normal page and not a pop-up, because `window.close` works only on pop-ups.



Figure 12-3. The Android browser is one of the few capable of opening pop-ups with a subwindow design; however, this ability was removed in Android 3.x.

So, if you can, avoid using pop-ups. If you really need one for some reason, open the window after an `onClick` event (avoid opening windows in the `onLoad` event or inside a timer callback) and remember that some mid-range and low-end devices will not show your window and will just replace current HTML with the pop-up destination.



The HTML specification from WHATWG (<http://whatwg.org/C>) includes the definition of a `dialog` element that can be used with an API to open modal pop-ups and customized dialogs. While we may see this element implemented in the future, today it is not included in the mobile browser world.

Some browsers will completely ignore a call to `window.close`, while some others, such as Safari on iOS, will close the current window and take the user back to the windows list (on iPhone or iPod devices).



A better alternative is to use a link with `target="_blank"`. This will have the same result on mobile devices as a `window.open` call, and it will work in every browser. If the browser doesn't support multiple windows, it will just replace the current one.

Focus and Scroll Management

You can set the focus to a clickable element (such as a form input, link, or button) using the `focus` function of every DOM element. The most helpful usage is for form input controls. The behavior varies in different mobile browsers. On some touch devices, focusing in a text box should automatically open the onscreen keyboard, and in some cursor-based browsers it will position the cursor over the element.



If the document the user is browsing is form-based, like a search page or a contact us page, it will be better for usability to automatically focus on the first text input. This reduces the amount of navigation the user has to do on the page.

On some devices, the global `window` object has a `scrollTo()` function that takes two parameters, `xPosition` and `yPosition`, specifying the position at the top-left corner of the screen to scroll to. On some devices (like the iPhone), using `scrollTo()` emulates the user's scrolling and hides the browser's toolbars, as if the user were scrolling with her fingers. So, for some browsers it is common to use the following code, which automatically hides the toolbars after the `onLoad` event:

```
window.scrollTo(0, 1);
```

This function can also be used to generate links to the top of the page, on compatible devices:

```
<a href="javascript:window.scrollTo(0, 1);">Go to Top</a>
```

This same behavior can also be applied without JavaScript, using anchors. We have already covered this technique and other similar hacks in Chapter 7.

Timers

JavaScript offers three kinds of timers: the well-known `setTimeout()` and `setInterval()`, and a new HTML5 timer, `requestAnimationFrame()`. The first one is executed once and the second one is executed every n milliseconds until it is cancelled using `clearInterval()`. The latest is not yet supported in every browser, and it's optimized for animations and games.

You can use timers for updating information from the server using Ajax every n seconds, for creating an animation, or for controlling the timeout of an operation.



In mobile browsers, you need to be especially careful about using timers because of the battery consumption. If you need to use many high-frequency timers at the same time, try to manage them using only one timer that will launch different behaviors from the same process.

The first question we need to ask ourselves is, what happens when our web page goes to the background because the user switches focus to another application (in multi-tasking operating systems) or opens or browses to another tab or window? Another problem is what happens when the phone goes to sleep (because of the user's inactivity while the script is executing). The behavior of timers can be a little tricky in these situations.

Yet another problem is that timers execute on the same thread as the main script. If our script is taking too much processor time (a normal situation with large scripts on low-end and mid-range devices), our timers will be delayed until some spare execution time is found.

If we use a low frequency for the timer (for example, 10 milliseconds), the timer will generally have problems meeting the timetable.



Remember that the JavaScript execution time depends a lot on the device hardware and the browser's engine. Even if two devices are running the same operating system, like Android, execution times can differ based on the device's processor.

Let's look at a simple example and see what happens normally:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Using Timers</title>
</head>

<body>

<script type="text/javascript">
var timer = setInterval(timerHit, 200);
var q = 0;
var lastTime = new Date().getTime();

function timerHit() {
    q++;
    var deltaTime = new Date().getTime() - lastTime;
    document.getElementById("content").innerHTML += q + ": " +
        deltaTime + "<br />";
    lastTime = new Date().getTime();

    // Generate some random delay
    var randomNumber = Math.floor(Math.random()*1000)+5000;
    for (var i=0; i<randomNumber; i++) {
        var a = new Array();
    }

    // We will run only 15 experiments
    if (q==15) {
        clearInterval(timer);
    }
}
</script>

<div id="content">
</div>

</body>
</html>
```

The real times are very different on different devices. On low-end and mid-range devices, if they work at all, the result is far from our 200 ms intention—some low-end devices don't even accept timers with a frequency of less than 1 second.

Remember that only a couple of browsers, such as Chrome for Android, maintain execution—including timers—when the device is locked, the tab is in the background, or the browser is minimized. Chrome will reduce the frequency of execution up to 1,000 ms when the page is not in the foreground.



The Gmail for Mobile team discovered some issues with timer behavior on mobile Safari and Android devices, and made the results public in the team blog (<http://www.mobilexweb.com/go/timers>). The conclusions are: for low-frequency timers (1 second or more), there are no performance issues, and you can add as many as you want; for high-frequency timers (for example, 100 ms), though, every new timer created makes the UI more sluggish. The preferred solution is to use only one high-frequency timer.

Animation timing

Timing control (<http://www.w3.org/TR/animation-timing/>) for script-based animations, also known as animation timing or as the function name `requestAnimationFrame()`, is a W3C draft specification that allows us to use an animation-optimized timer instead of `setTimeout()`.

When using this technique, we don't specify the frequency interval; instead, the browser will determine the best time to update the animation regarding the current scenario. Therefore, the browser specifies the frame rate, and it will pause the animation if it believes the animation should be paused (such as in a background operation).

The standard API is simple: we have a global `window.requestAnimationFrame()` function that receives only the function to execute and returns an ID that we can store, to use with `cancelAnimationFrame()` as we would with `clearTimeout()`. If we want the animation to continue to a next frame we must call `requestAnimationFrame()` in the animation function; therefore, there is no easy way to use this technique with anonymous inline functions.



`requestAnimationFrame()` works similarly to `setTimeout()`, not `setInterval()`, as it doesn't execute many times automatically.

To start an animation frame, we can use code like this:

```
var animationId = requestAnimationFrame(animate);

function animate() {
  // animate
  if (!endAnimation) {
    requestAnimationFrame(animate);
  }
}
```

And to stop the animation:

```
cancelAnimationFrame(animationId);
```

At the time of this writing, Internet Explorer 10 is the only browser supporting this API without any prefix. Firefox supports it with a `moz` prefix and Safari from version 6.0, BlackBerry from version 10, the webOS browser for the TouchPad, and Chrome for Android support it using a `webkit` prefix.

On WebKit-based browsers, `requestAnimationFrame()` accepts a second optional argument: a DOM element that is the one that visually bounds the animation.

If we want to encapsulate all the prefixed versions (and we are sure everyone is using the same syntax), we can use the following code:

```
(function() {  
    var requestAnimationFrame = window.requestAnimationFrame ||  
                                window.mozRequestAnimationFrame ||  
                                window.webkitRequestAnimationFrame ||  
                                window.msRequestAnimationFrame;  
    window.requestAnimationFrame = requestAnimationFrame;  
})();
```

Paul Irish (<http://paulirish.com>) has created a nice polyfill so we can use the Animation Timing API on every browser, even falling back to `setInterval()` in noncompatible browsers. It's available at GitHub (<https://gist.github.com/1579671>).

Changing the Title

In desktop web applications, it is common to change the title dynamically to alert the users of a change in the page, when updates are made in an Ajax application, or simply as an animation (please, don't do this!).

In mobile browsers, this isn't such a good idea, for the following reasons:

- Many browsers don't even display the title.
- If the user is working with many tabs at the same time, dynamically changing the title won't be useful because your web page will be frozen when it is in the background on most browsers.
- Animations in the title can be annoying in a mobile browser.

Cookie Management

Cookies are a great solution for the problem of statelessness in HTTP. They work in all devices and browsers. This is good. The bad thing is that the lifetime of a cookie can be shorter in the mobile ecosystem than in the desktop world, especially in low-end and mid-range devices because of the lack of memory storage.



It is recommended to maintain cookies' values at below 2 KB for the best compatibility in mobile devices. Be careful when using these server-based techniques in native web apps, where there is no HTTP protocol in the main HTML and script files.

Cookies are normally stored and read by the server, but JavaScript also allows us to read and write them as a client-side storage mechanism.

Event Handling

One of the most frequently used features of JavaScript is event handling, whether we define it inside the HTML document or by using code. Let's see how mobile browsers work with this way to execute script code.

Managing events

We can define event handling in scripts using the following methods:

- Using HTML attributes, like `onclick="alert('sample')"`
- Using the JavaScript object property, `element.onclick = function() {}`
- Using the DOM `addEventListener()` method



Before version 9, Internet Explorer used the `attachEvent` property of the element instead of the DOM `addEventListener()` method.

Load and unload events

The famous load event is available for any HTML element, but it is best used in the body element (the document JavaScript object). We'll test compatibility in different types of elements here.

The unload event is less famous. In theory it should work for every element, but again the most useful usage is applied to the body element (document object) to detect when the user is navigating away from our document.

In modern browsers, the unload event does not work as we might want (I remember many battles against the unload event when a new pop-up was opened every time I closed one), and it has been replaced by the nonstandard `beforeunload`. The `beforeunload` event is useful for alerting the user about unfinished work so he doesn't lose any changes he's made on the page before going back or browsing to another URL. To do this, it is generally used with a confirm dialog. Unfortunately, `beforeunload` is not

so compatible with mobile browsers, so we should store information frequently instead of relying on a confirmation dialog on the `beforeunload` event.

Click events

The `click` event is the most-used event on the Web. In mobile sites, we have to test it to see where it can best be used. We know that there are focus-based, touch-based, and cursor-based browsers. The last ones are the simplest for `click` events: every time the user moves the cursor arrow and then presses Fire or any other similar key, a `click` event is generated. In focus-based browsers, it is recommended to use the `click` event only in clickable elements, such as links or buttons, because the focus may not be active on other elements (such as `div`, `p`, or `li` elements).

For touch devices, the behavior is simple, too: every touch (finger- or stylus-based) is transferred as a `click` over the screen.



If the user is using a finger to touch the screen, you need to be aware that the `click` coordinates can change during the touch (depending on how the user presses the screen), and the precision will not be good. Use big areas as clickable ones. Also, for touch devices we can use touch or pointer events, as we will see in Chapter 15.

Double tap. In Chapter 15, we will discuss touch and pointer events, but for now, let's talk about how to emulate a double-click action.

On touch devices, if you want to detect a double-tap gesture, you shouldn't use the nonstandard `ondblclick` event; in most cases it will not work and it will also fire an `onclick`. The best solution (also compatible with nontouch devices) is to implement a tap-double tap detection pattern using the following code sample:

```
var doubletapDeltaTime_ = 700;
var doubletap1Function_ = null;
var doubletap2Function_ = null;
var doubletapTimer = null;

function tap(singleTapFunc, doubleTapFunc) {
  if (doubletapTimer==null) {
    // First tap, we wait X ms to the second tap
    doubletapTimer_ = setTimeout(doubletapTimeout_,
      doubletapDeltaTime_);
    doubletap1Function_ = singleTapFunc;
    doubletap2Function_ = doubleTapFunc;
  } else {
    // Second tap
    clearTimeout(doubletapTimer);
    doubletapTimer_ = null;
    doubletap2Function_();
  }
}
```

```

    }
}

function doubletapTimeout() {
    // Wait for second tap timeout
    doubletap1Function_();
    doubleTapTimer_ = null;
}

```

We can use the previous library like this:

```

```

supposing `tapOnce` and `tapTwice` are two previously declared global functions.



In general, in a nonclickable element no events will be generated, while in clickable elements events are fired in the order `onmouseover`, `onmousedown`, `onmouseup`, `onclick`.

Alternatively, we can use it from JavaScript as follows:

```

element.onclick = function() {
    tap(
        function() {
            // This is the code for the first tap
        },
        function() {
            // This is the code for the second tap
        }
    );
}

```



Remember that implementing touch-and-hold (or long press) handling can cause problems in some touch browsers because the browser is already capturing this event for contextual menus. You can apply it only in text blocks with `user-select` disabled. We may also have problems with the double tap if we have zoom enabled, as the browser may treat it as a zoom in/out action.

Focusable and form events

Support for the `focus`, `blur`, `change`, and `submit` (only for forms) events in different mobile browsers is good enough to use them without any worries.

Over events

The over events include `mouseover` and `mouseout` and are typically used for creating a hover effect when the cursor is over an element. Usage of these events in mobile websites

is discouraged for must-have features, because they will only work in cursor-based browsers. Touch and focus devices don't have an "over" state; it should be replaced by an active state or a focus one for focus-based browsers.



Safari on iOS also supports the `mousewheel` event when the user is scrolling the element using two fingers at the same time.

Resizing, scrolling, and orientation change events

When the user activates scrolling over the document, some browsers fire the `scroll` event from the document as a whole. The main problem is that when the scroll has momentum the event is being fired only when the scroll momentum ends, and not continuously, as in desktop browsers. You can find out more about this problem and some workarounds at TJ Vantoll's website (<http://bit.ly/ZCsXyP>).

Some browsers also support the `onresize` event, which fires when the window size is changed. Users cannot resize mobile browser windows in the way they can resize desktop application windows, but a resize can be generated if the orientation of the device changes from portrait to landscape or vice versa.

Most mobile browsers support the `orientationchange` window event and the `orientation` property of the window global object. This property has a value of 0 in portrait mode, 90 in landscape mode, and -90 in inverse landscape mode. Tablets also support the 180 value for inverse portrait mode, which is not possible on phones. We can use this value to make changes in the DOM, or use a body class pattern:

```
if (window.orientationchange) {
  window.addEventListener("orientationchange", function() {
    var orientation = window.orientation;
    switch(orientation) {
      case 0:    // Portrait
        break;
      case 90:   // Landscape to the left
        break;
      case -90:  // Landscape to the right
        break;
      case 180:  // Up side down portrait
        break;
    }
  }, false);
}
```

If the device doesn't support the `orientationchange` event, we can query on `resize`. That is, we can detect the change using the following code:

```
if (window.orientationchange==undefined) {
  window.addEventListener("resize", function() {
```

```

        if (screen.width>screen.height) {
            // Landscape
        } else {
            // Portrait
        }, false);
    }
}

```

Key events

Key events—keypress, keyup, and keydown—allow us to detect keypresses over the whole page (body) or in one element (generally, a text input). On compatible mobile devices, this can be useful for many situations:

- To provide keyboard shortcuts
- To provide navigation or movement in a game or application
- To enable form submission on Enter or another keypress
- To disallow some characters in a text input

If we are going to prevent a key from being used, we should be very careful. Remember that devices can have very different keyboards. Some devices have only virtual keyboards, some numeric, and some QWERTY, and key code management across platforms can be a little tricky.



If the device has a QWERTY keyboard we can also detect some modifier keys (if they exist), like Ctrl, Alt, or Shift, using the event properties.

A simple test for getting key codes can be created using the following code:

```

<script type="text/javascript">
window.addEventListener("keyup", function(event) {
    // charCode depends on modifiers (such as shift); keyCode does not
    var code = event.keyCode ? event.keyCode : event.charCode;
    alert("code: " + code + " - ASCII value: " + String.fromCharCode(code));
}, false);
</script>

```

Useful keys for some devices. In Safari on iOS, while the focus is inside a text input with the keyboard visible onscreen, we can capture every key pressed using only keyCodes. Table 12-2 shows some important codes.

Table 12-2. Useful keyCode in Safari

Key	keyCode
Backspace/Del	127
Enter	10
Space	32

There are Android and webOS (Palm) devices with physical keyboards, and others without them. The possible special key values for all these devices are shown in Table 12-3.

Table 12-3. Useful keyCode in Android and webOS

Key	keyCode
Backspace/Del	8
Enter	13
Space	32



Remember that most touch devices don't have a physical keyboard, and there is no way to open the virtual keyboard whenever we want; it is only opened when some typing-enabled element is in focus, such as a text input or an editable element. Some devices allow the usage of an external keyboard, but the ability to read those keys without an editable element varies per browser.

The Nokia N97 has a full QWERTY keyboard, but the letters don't provide the correct ASCII values unless the user presses the Shift key at the same time. For example, the H and I keys provide the same keyCode (56) but different charCodes. The default Unicode values for the charCodes are the numeric or symbol values of the keys (typically used with the Sym key). If the user is using the onscreen keyboard (only available as a pop-up window when a form has focus), every character typed is delivered (regardless of whether it was entered on the numeric keyboard, by touch recognition, or by predictive text). Table 12-4 shows the common codes.

Table 12-4. Useful `keyCode`s and `charCodes` for Symbian touch devices

Key	keyCode	charCode
Backspace	8	8
Enter	13	13
Space	32	32
Up	38	63497
Down	40	63498
Left	37	63495
Right	39	63496
Fire	N/A	63557



Even if we can capture keypresses, remember that special keys (Menu, Call, End, Volume) are generally out of our scope as web developers. We cannot detect those keys.

Older Symbian 3rd edition devices (including the Nokia N95, E61, and so on) are non-touch devices with numeric keypads. The few keys we can capture on such devices are shown in Table 12-5.

Table 12-5. Useful `keyCode`s and `charCodes` for Symbian 3rd edition devices

Key	keyCode	charCode
Clear	8	8
Send	N/A	63586
Cursor and Fire	N/A	N/A

Preventing default behavior

For almost every event, we can prevent the default behavior by using the `event.preventDefault()` method or capturing the event and returning `false`. This is commonly done with the `submit` event, to cancel the submission when something doesn't validate, or to cancel a link. For example:

```
<a href="news.html" onclick="news();return false">Go to news</a>
```

The preceding code is a standard link to *news.html*, but if JavaScript is supported we can capture the `click` event, call a local function (that can get the news by Ajax), and cancel the normal behavior of the link by returning `false`. This avoids a page load and reduces network traffic.

We can also prevent a key from being used by cancelling the `keyup` event. This feature must be used very, very carefully, and only on tested devices.

JavaScript Libraries

The life of a JavaScript programmer has changed radically since 2006, with the appearance of Ajax and hundreds of libraries that help us work better with this language. Many of these libraries modify existing behaviors or add completely new behaviors to the language, effectively creating new languages inside (or over) JavaScript.

If the libraries are based on JavaScript, and mobile browsers support JavaScript, why do we care? The answer is that many of these libraries rely on some not-so-clear things in the standard, and while they have been prepared and tested on well-known desktop browsers (Internet Explorer, Firefox, Safari, Chrome, Opera), they have not been tested on all the mobile browsers.

That is the first reason why we need to be careful about using big JavaScript libraries. The second (no less important) reason is the impact on download and execution times, as well as battery consumption. As mentioned earlier, these libraries modify the language and the behavior of standard objects, and even if we don't use any (or very little) of a library's code, the library will need to load itself completely, which takes time. This can lead to performance problems in some browsers, so we are going to test the time that typical libraries take to initialize themselves on mobile browsers.



Some libraries, in their uncompressed form, are larger than 600 KB. We need to be very careful about performance when using that code, as it will increase network traffic, memory consumption, and execution time. If you can, avoid those big libraries, or use only the code you need.

My first advice is that you should avoid these libraries if you can. If you cannot avoid them, use them only for smartphones, and be aware that some features and plug-ins may not work properly. Look for a way to download only the part of the library that you really need to use; some libraries, such as jQuery Mobile, offer a download builder that creates a customized package for your needs.

When we analyze framework or library usage, we need to understand that the first big problem is just loading and parsing the JavaScript file, even if we are not using it yet. To parse the jQuery core file the browser needs to get and parse around 100 KB of JavaScript code, even if we won't be using it for a while. Downloading and parsing big files can harm your website's performance and use up battery power, so use them with care and only when you really need them.

Just to give you an idea, on some old BlackBerry 4.x devices, the browser needs 8 seconds just to parse the jQuery core file, during which time it shows a "loading" screen and no user interaction with the website is possible.



If you use jQuery just because you don't want to type `document.getElementById()` or `document.querySelector()`, go now and remove all jQuery references in your mobile websites.

Vanilla JS

Vanilla JS (<http://vanilla-js.com>) is a fast, lightweight, cross-platform framework for building incredible, powerful JavaScript applications. You can create your own modules, including support for animations, regular expressions, closures, array libraries, the DOM, Ajax, and much more. Go now and check out the website: on the site you can compare its performance with jQuery, Prototype, Dojo, and MooTools and see how much faster it is.

Really, go. I'll wait here.

If you've just gotten back from checking out the website, maybe you have already realized that this is an ironic library. Vanilla JS doesn't exist, because it's just plain JavaScript, available by default in every browser! The moral of this story is that you don't need a library for a lot of things, and if you avoid using them your users will be much happier.

Mobile Libraries

Sometimes libraries are useful, though, and the good news is that many developers have released alternative libraries that are geared for mobile devices and are lighter than the classic ones. There are also full UI frameworks for mobile application development (mostly prepared for iPhone and Android devices) that we will cover later, like jQuery Mobile, iUI, iWebKit, and Sencha Touch. These frameworks will take care of the visualization, events, and interaction of our websites.

There are also other libraries that can replace jQuery and the others on mobile devices. They are very light libraries that provide basic DOM, event, and Ajax support.

Zepto.js

Zepto.js (<http://zeptojs.com>) is a minimized JavaScript library optimized for performance. Its goal is to provide a 5–10 KB modular library that downloads and executes fast, with a jQuery-like syntax. With its inclusion with the HTML5 Mobile Boilerplate (<http://html5boilerplate.com/mobile/>) template, Zepto.js has become the most popular replacement for jQuery in mobile web development.

As a jQuery-based library it uses the same `$` query selector and most of the typical jQuery functions, such as the DOM, Ajax, forms, and event methods.



Zepto includes a Touch module prepared for mobile devices that we can use to detect some basic gestures, such as `tap`, `doubleTap`, `long Tap`, and `swipe` variations.

baseJS

baseJS (<http://paularmstrongdesigns.com/projects/basejs>) is a lightweight library (8 KB) compatible with Safari on iOS and other WebKit-based browsers. It has only been fully tested on Safari, from iOS 1.0 to 3.0.

baseJS provides a selector similar to jQuery's `$(selector)`, and some similar methods, like `each()`, `addClass()`, `hasClass()`, `removeClass()`, `toggleClass()`, `getXY()`, `fire()`, and some Ajax methods.

jQMobi

jQMobi (<http://www.jqmobi.com>) is a query selector library optimized for mobile devices that takes 5 KB of JavaScript code using a jQuery syntax. Based on Zepto and jQuery, this library focuses on ways to reduce the API to its most useful features and offers a fast version of a similar API. It includes basic DOM and Ajax management through the `$` object and chained functions.

QuoJS

Quo (<http://quojs.tapquo.com>) is a micro JavaScript library (14 KB) optimized for mobile devices that includes DOM, event, and Ajax functionality and a good quantity of touch gesture event handlers, such as `pinch out`, `drag`, `rotate`, `hold`, `swipe`, and `two-finger tap`.

XUI

XUI (<http://xuijs.com>) is a simple JavaScript framework for building mobile websites that takes up only 4.2 KB compressed. It is available for free and has been fully tested on WebKit-based browsers and Opera Mobile. The developers are working on adding support for IE Mobile and BlackBerry.

XUI is also similar to jQuery, but it is more powerful than baseJS. XUI uses `x$` as the main selector object and includes the methods listed in Table 12-6.

Table 12-6. XUI common methods for a selector query

Method	Description
<code>html(<i>code</i>)</code> or <code>html(<i>location</i>, <i>code</i>)</code>	Defines the inner HTML (or other location, using the second option) of the elements retrieved by the selector query. The <i>location</i> is a string and can be one of the following: <code>inner</code> , <code>outer</code> , <code>top</code> , <code>bottom</code> , <code>before</code> , <code>after</code> , or <code>remove</code> .

Method	Description
<code>on(event, function)</code>	Registers an event listener. The event name can also be used directly as the method name (e.g., <code>click</code> rather than <code>on('click')</code>). The events compatible are: <code>click</code> , <code>load</code> , <code>touchstart</code> , <code>touchmove</code> , <code>touchend</code> , <code>touchcancel</code> , <code>gestures tart</code> , <code>gesturechange</code> , <code>gestureend</code> , and <code>orientationchange</code> .
<code>setStyle(property, value)</code>	Defines a CSS style.
<code>getStyle(property, optional_callback)</code>	Reads the value of a property. If the selector has multiple elements, the callback will be fired.
<code>addClass(class_name)</code>	Adds a class to the elements.
<code>removeClass(class_name)</code>	Removes a class from the elements.
<code>css(object)</code>	Defines CSS styles using a JSON-style object having properties with values.
<code>tween(object)</code>	Animates one or more CSS properties from one value to another defined in the object.

So, for example, we can capture an onclick for buttons with a class with the following code:

```
$('#input.button').on('click', function(e){ alert('Ouch!') });
```

or with code like this, chaining the methods *à la* jQuery:

```
$('#input.button').click(function(e){ alert('Ouch!') })
    .html('Press Me! ').css({color: 'blue'});
```

For Ajax, XUI provides global `xhr()` and `xhrjson()` functions to create requests with options.

UI Frameworks

There are dozens of JavaScript UI libraries for implementing rich controls in desktop websites. The question is: do they work on mobile devices? Usually the answer is no, but even if it's yes it's best to avoid them. They are not usually optimized for touch or focus usage, and the UI will be slow and sluggish.

Many UI libraries have appeared on the market in the past few years to facilitate mobile development. Unfortunately, most of them were designed for specific platforms, but with minor changes or incompatibilities many of them should work on any device.

The browser with the most specific platform is Safari on iOS, followed by the Android browser (and other WebKit-based browsers). Lots of libraries allow us to create rich applications emulating native control behaviors for iOS.

Common iOS-based UI libraries (that also work on Android and may work on other WebKit-based browsers) include:

- iUI (<http://code.google.com/p/iui>)
- jQTouch (<http://jqtouch.com>)

- iWebKit (<http://iwebkit.net>)
- jqUI (<http://www.jqmobi.com>)
- WebApp.Net (<http://webapp-net.com>)
- ciUI (<http://code.google.com/p/ciui-dev>), a C-NET alternative to iUI
- Universal iPhone UI Kit (<http://code.google.com/p/iphone-universal>)

Targeting multiplatform compatibility, we can mention:

- jQuery Mobile (<http://www.jquerymobile.com>), supporting all smartphone and tablet platforms
- Sencha Touch (<http://www.sencha.com/products/touch>), supporting iOS, Android, modern BlackBerry, and Windows Phone 8 platforms
- Lingo (<http://lingo.tapquo.com>), supporting smartphone, tablet, and smart TV platforms

BlackBerry offers the BBUI.js library (<https://github.com/blackberry/bbUI.js>) that gives a native-like UI in web apps for BlackBerry 7, PlayBook, and BB10 automatically. It's optimized for native web apps using the WebWorks packager.

Sencha Touch

Sencha Touch (<http://www.sencha.com/products/touch>) is a JavaScript-based framework for mobile app development. It has a free commercial license and it is compatible with iOS, Android, BlackBerry 7/10, and Windows Phone 8. It offers a complete multiplatform development environment including the UI, application data management, and binding. The Sencha SDK, from the same company, allows us to make native web apps, compiling the app directly from an IDE.

Sencha Touch includes more than 50 components (see Figure 12-4) and a built-in MVC (Model-View-Controller) system written completely in JavaScript. When using Sencha Touch, our HTML files will consist of just a couple of JavaScript include files and an empty body element. Everything will be instantiated from JavaScript code and rendered by the framework.

A basic “Hello World” application in Sencha Touch will look like this:

```
Ext.application({
    name: 'Sencha Touch Demo',

    launch: function() {
        Ext.create("Ext.tab.Panel", {
            fullscreen: true,
            items: [
                {
                    title: 'Hello World',
```

```

        iconCls: 'home',
        html: 'Welcome to Sencha Touch'
    }
  });
});

```

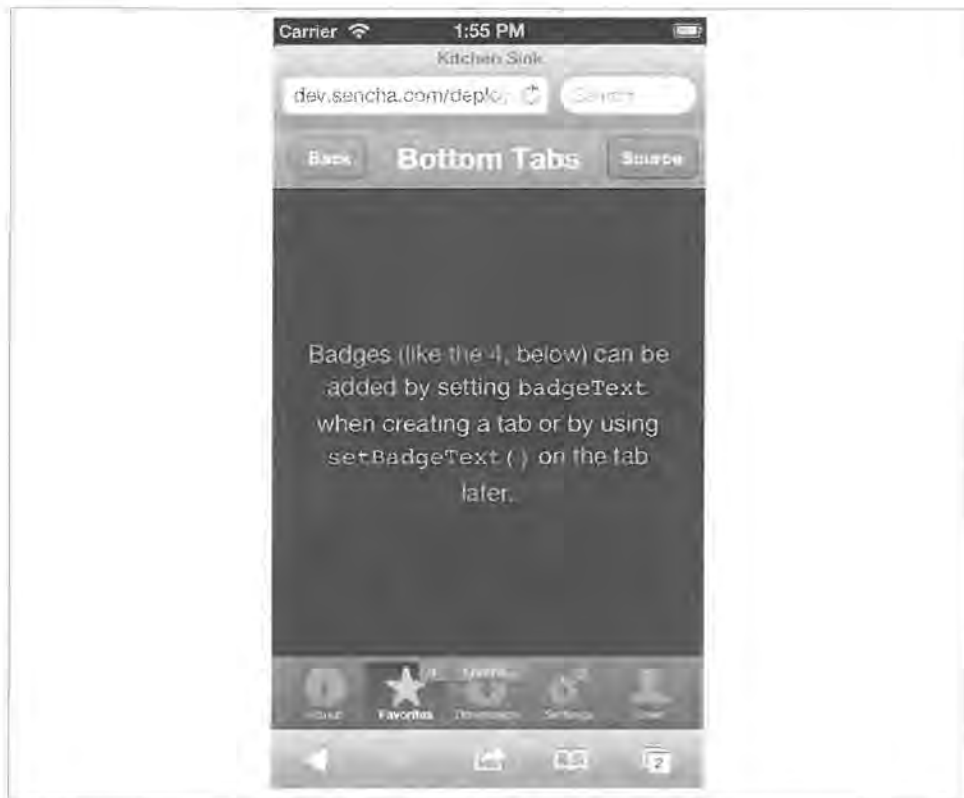


Figure 12-4. Sencha Touch emulates the native UI in HTML5 browsers using an MVC architecture.

If you want to learn more about this framework, pick up the book *Sencha Touch 2: Up and Running*, by Adrián Kosmaczewski (O'Reilly).

jQuery Mobile

jQuery Mobile (<http://jquerymobile.com>) is a cross-platform UI framework optimized for touch smartphones. jQuery Mobile is not a mobile replacement for the jQuery core, like Zepto.js or jQMobi; jQuery Mobile works over jQuery and it manages only the user interface of our web apps.

I've written a whole book about this framework, called *jQuery Mobile: Up and Running* (O'Reilly).

jQuery Mobile is compatible with more than 10 different platforms, including all the modern smartphone and tablet browsers; it's open source and is based on semantic HTML5 markup, so it doesn't require any JavaScript coding.

jQuery Mobile manages a series of rich controls, as well as page navigation and transitions using CSS3 animations. It is compatible with Apache Cordova/PhoneGap apps.

A typical jQuery Mobile document looks like the following code, which we can see rendered in Figure 12-5:

```
<div data-role="page">
  <div data-role="header" data-position="fixed">
    <h1>Hello World</h1>
    <a href="#">Action</a>
  </div>

  <div data-role="content">
    <ul data-role="listview" data-filter="true">
      <li>Option 1
      <li>Option 2
      <li>Option 3
      <li>Option 4
    </ul>
  </div>

  <div data-role="footer" data-position="fixed" data-theme="e">
    <h4>O'Reilly Media</h4>
  </div>
</div>
```

Enyo

Enyo (<http://enyojs.com>) is a cross-device (desktop, tablet, and phone), cross-platform web app UI framework. The framework originated with the HP Palm team from webOS, but after the open sourcing of that platform (see Chapter 2 for more information) the Enyo effort was converted to a multiplatform solution, including support for iOS, Android, Firefox, Chrome, Windows Phone, BlackBerry, and more.

Enyo is modular and component-based and the basic core is only 25 KB zipped, making it a mobile-optimized framework. It has a range of useful widgets, including image carousels and date pickers, and we can encapsulate our own controls really easily.

I've written a whole book about this framework, called *jQuery Mobile: Up and Running* (O'Reilly).

jQuery Mobile is compatible with more than 10 different platforms, including all the modern smartphone and tablet browsers; it's open source and is based on semantic HTML5 markup, so it doesn't require any JavaScript coding.

jQuery Mobile manages a series of rich controls, as well as page navigation and transitions using CSS3 animations. It is compatible with Apache Cordova/PhoneGap apps.

A typical jQuery Mobile document looks like the following code, which we can see rendered in Figure 12-5:

```
<div data-role="page">
  <div data-role="header" data-position="fixed">
    <h1>Hello World</h1>
    <a href="#">Action</a>
  </div>

  <div data-role="content">
    <ul data-role="listview" data-filter="true">
      <li>Option 1
      <li>Option 2
      <li>Option 3
      <li>Option 4
    </ul>
  </div>

  <div data-role="footer" data-position="fixed" data-theme="e">
    <h4>O'Reilly Media</h4>
  </div>
</div>
```

Enyo

Enyo (<http://enyojs.com>) is a cross-device (desktop, tablet, and phone), cross-platform web app UI framework. The framework originated with the HP Palm team from webOS, but after the open sourcing of that platform (see Chapter 2 for more information) the Enyo effort was converted to a multiplatform solution, including support for iOS, Android, Firefox, Chrome, Windows Phone, BlackBerry, and more.

Enyo is modular and component-based and the basic core is only 25 KB zipped, making it a mobile-optimized framework. It has a range of useful widgets, including image carousels and date pickers, and we can encapsulate our own controls really easily.

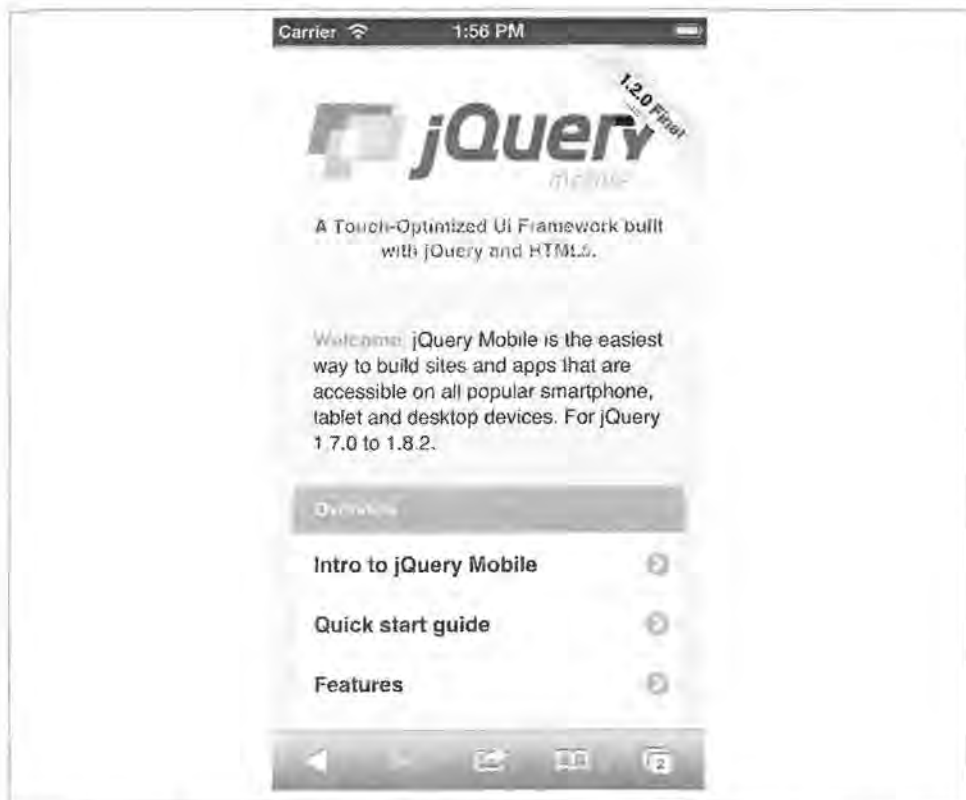


Figure 12-5. jQuery Mobile offers a touch-optimized UI for almost every platform available on the market, using semantic HTML5 code.

Montage

Montage (<http://montagejs.org>) is a complete web app development framework, including a client-side JavaScript library and a server-side Node.js library. One of the big advantages of Montage is the usage of a two-way data binding mechanism using custom HTML5 data, which allows us to simplify the development process and the business logic of our apps. It also includes components and templates, data persistence, and event management.

Montage is an open source project created by Motorola Mobility, now owned by Google. One of the key targets of Montage is Apache Flex (formerly Adobe Flex) developers trying to move into the HTML5 world.

iUI

iUI was one of the first libraries developed for iPhone-style application development. It was developed by Joe Hewitt (<http://www.joehewitt.com>), author of the popular Firefox plug-in FireBug.

iUI is a very simple-to-use, nonintrusive JavaScript library, including CSS and assets (generally backgrounds) that emulate the native iPhone application controls, animations, and application workflow.

The main features are:

- No JavaScript coding required
- Extends the behavior of standard HTML markup
- Overrides links and forms with Ajax requests
- Updates the page with smooth iPhone transitions with no code

The library uses the master-detail navigation method, copied from the UITableView control from Cocoa Touch. UITableView is a native iPhone control.

The main structure of an iUI website is just a simple HTML file that includes CSS and JavaScript files and contains a div for the top toolbar and one ul for each “list page” we want to show:

```
<!-- Leave this toolbar div with no changes -->
<div class="toolbar">
  <h1 id="pageTitle"></h1>
  <a id="backButton" class="button" href="#"></a>
</div>
<!-- This is the home screen -->
<ul id="home" title="Home Page" selected="true">
  <li><a href="#first">First</a></li>
  <li><a href="#second">Second</a></li>
  <li><a href="#third">Third</a></li>
  <li><a href="ajax.html">Loaded by AJAX</a></li>
  <li><a href="more.html" target="_replace">More...</a></li>
</ul>
<ul id="first" title="Other Screen">
  ... other options ...
</ul>
... other pages
```

iUI uses the child element with `selected="true"` as the home screen when the page loads the first time, and every link with a hash (#) is a link to another screen that loads on the same page and is identified by the ID after the hash, *à la* WML card.

The toolbar is always present and the Back button works automatically, restoring the previous screen's title and using a smooth swipe animation to go back.

To reference an external file, we can use a normal link: it will load via Ajax and be inserted below the toolbar. The external file should only have a `ul` or `div` element without any other root tag.

We can also implement an in-site pagination design pattern using `target="_replace"`. This will load the href document using Ajax and insert its contents where the `li` with the replace link is defined. This document should only have `lis` without any other root element, and it should end with another replace link if there are more elements to paginate.

Other advanced features that iUI supports include:

- Modal dialog pop-ups
- Form designs
- Emulation of native form controls
- Stripped tables
- Right (>) toolbar button

You can find more samples of iUI usage at www.mobilexweb.com/go/iui.

jQTouch

jQTouch is a jQuery plug-in for iPhone web development that produces similar results to iUI, but with more powerful graphics and animations. It is also unobtrusive, but it does require some JavaScript code to initialize the page.

With some visual differences, it also works on the Android and webOS browsers.

The supported features include:

- Native WebKit animations using jQuery methods
- Callback events
- Flexible themes
- Swipe detection
- Extensions: floaty bar, geolocation, offline capability
- Visual controls
- Animations

To use jQTouch we need to first load jQuery 1.3.2 and then the jQTouch script, with two CSS files: the base file (*jqtouch.min.css*) and the visual theme we want to load. For example:

```

<script type="text/javascript" src="http://www.google.com/jsapi">
</script>
<!-- We load jQuery using the Google Ajax API -->
<script type="text/javascript">google.load("jquery", "1.3.2");</script>
<script type="text/javascript" src="jqtouch/jqtouch.min.js"></script>
<style type="text/css" media="screen">
    @import "jqtouch/jqtouch.min.css";
</style>
<style type="text/css" media="screen">
    @import "themes/jqt/theme.min.css";
</style>

```

To initialize the page, we can use:

```

<script type="text/javascript">
$.jqTouch({
    icon: 'iphone-icon.png',
    preloadImages: [
        'themes/jqt/img/chevron_white.png',
        'themes/jqt/img/bg_row_select.gif',
        'themes/jqt/img/back_button_clicked.png',
        'themes/jqt/img/button_clicked.png'
    ]
});
</script>

```

There are dozens of properties we can define at the initialization. You can find a list at code.google.com/p/jqtouch/w.

As in iUI, the main markup is done using top-level elements (in this case, `divs`) with `ids` and links with hashes for linking in the same document. For example:

```

<div id="home" class="current">
    <div class="toolbar">
        <h1>jqTouch</h1>
        <a class="button slideup" id="infoButton" href="#about">
            About
        </a>
    </div>
    <ul class="rounded">
        <li class="arrow"><a href="#ui">User Interface</a>
            <small class="counter">4</small></li>
        <li class="arrow"><a href="#animations">Animations</a>
            <small class="counter">8</small></li>
        <li class="arrow"><a href="#ajax">Ajax</a>
            <small class="counter">3</small></li>
        <li class="arrow"><a href="#callbacks">Callback Events</a>
            <small class="counter">3</small></li>
        <li class="arrow"><a href="#extensions">Extensions</a>
            <small class="counter">4</small></li>
        <li class="arrow"><a href="#demos">Demos</a>
            <small class="counter">2</small></li>
    </ul>

```

```

<h2>External Links</h2>
<ul class="rounded">
  <li class="forward"><a href="http://www.jqtouch.com/"
    target="_blank">Homepage</a></li>
  <li class="forward"><a href="http://www.twitter.com/jqtouch"
    target="_blank">Twitter</a></li>
  <li class="forward"><a
    href="http://code.google.com/p/jqtouch/w/list"
    target="_blank">Google Code</a></li>
</ul>
</div>

```

JavaScript Mobile UI Patterns

As we saw in Chapter 5, mobile devices have had to develop alternate paths for handling many common tasks. In this section we'll take a closer look at a few mobile-specific UI patterns.

Clear text box buttons

In their native UIs, touch devices have added a very nice feature to text boxes: the possibility of clearing all the text by touching a small X at the righthand side of the box (as shown in Figure 12-6). This is especially useful because of the lack of a keyboard on these devices. We can emulate this UI pattern easily by combining an image (or, as we'll see later, a canvas tag) and a little JavaScript code.



Figure 12-6. You can see this pattern implemented in the Yahoo! website for touch devices like the iPhone.

To implement the clear button, we can use a 20×20-pixel image (great for inline images in compatible devices) with the following CSS style. The image can be shown as a `div` with a background image from the beginning, or only when the user starts typing. It is important to add a right padding to the input box so the X is not overlapped by text:

```
<style type="text/css">
div.clearx {
    background: transparent url('clearx.png') no-repeat right;
    height: 20px;
    width: 20px;
    margin-top: -26px;
    position: absolute;
    left: 235px;
}
input.clearx {
    padding: 2px 40px 2px 10px;
    width: 200px;
    height: 24px;
}
</style>
```

The HTML should look like this:

```
<input type="text" id="search" placeholder="Enter your search"
      class="clearx" />
<div class="clearx"
      onclick="document.getElementById('search').value=''"></div>
```

Autogrowing textareas

This UI pattern was created by the Google Mobile team and is currently used in Gmail. The problem is that if we have a large amount of text in a textarea, scrolling inside it is very painful in some browsers (Safari on iOS is one of them). The solution is to grow the textarea to fit the contents, so the user can use the normal page scrolling instead of the textarea's.



All of these JavaScript UI patterns can be created using a nonintrusive, object-oriented approach with a little JavaScript work.

We can capture the `onkeyup` event and grow the textarea if necessary. We also need to capture `onchange`, because pasting in iOS doesn't generate an `onkeyup` event.

The complete solution is available at www.mobilexcode.com/go/autogrowing. The code, borrowed from the Google Code Blog with a few changes, is:

```
<script>
// Value of the line-height CSS property for the textarea.
```

```

var TEXTAREA_LINE_HEIGHT = 13;

function grow(event) {
    var textarea = event.target;
    var newHeight = textarea.scrollHeight;
    var currentHeight = textarea.clientHeight;
    if (newHeight > currentHeight) {
        textarea.style.height = newHeight + 5 * TEXTAREA_LINE_HEIGHT +
            'px';
    }
}
</script>
<textarea onkeyup="grow(event);" onchange="grow(event);" >
</textarea>

```



The Google Mobile team is doing a great job with mobile web UI patterns and optimizations, and it releases all the tips to the public in the team blog (<http://googlecode.blogspot.com>).

Floating bars

Scrolling a large mobile web page just to access a button or a link at the top of the document can be very painful. A floating bar is a great solution for avoiding this problem when `position: fixed` (described in the “Display Properties” on page 404 section in Chapter 11) is not available. A floating bar is just a full toolbar, a drop-down menu, or a mixture of both that always remains at the top (or bottom) of the page when the user scrolls the content.

It is not suitable for focus-based browsers, because there will be usability issues when the user is tabbing between links.



We need to create a custom floating bar solution only when `position: fixed` is not available. If this feature is available, we should go with it.

For floating bars to work, the browser needs to support the `scroll` event. If the browser supports this event, the toolbar moving can be done automatically, using a smooth animation on some browsers. You can decide whether to have the floating bar appear at the beginning of the navigation or only after scrolling.

The steps to create a floating bar (also known as a *floaty bar*) are:

1. Create a `div` with the content of the floaty bar.
2. Define it as hidden off the screen with negative top values.

3. Define a WebKit transition animation (this will work only on compatible devices).
4. Capture `onscroll`.
5. If the value of `window.scrollTop` (the top position of the scroll) is near zero, hide or move the div off the screen; if not, move the div (changing the `top` value) to the `scrollTop` position.



For Safari on iOS, there is a solution that will have better performance: instead of using a transition animation and changing the `top` value, we can use the `translateY()` function and do a transformation animation. Transformations use hardware implementations for improved performance.

The following code produces a floaty bar with animation for WebKit-based browsers:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Mobile Web Test Suite</title>
<style>

  p {
    font-size: xx-large;
  }

  #floaty {
    width: 200px;
    text-align: center;
    border: 2px solid red;
    -webkit-border-radius: 5px;
    background-color: silver;
    right: 0px;
    position: absolute;
    top: -50px;
    -webkit-transition: top 0.2s ease-out;
  }

</style>
<script type="text/javascript">
window.onscroll = function() {
  var floaty = document.getElementById("floaty");
  if (window.scrollTop<10) {
    // It is near the top, so we can hide the floaty bar
    floaty.style.top = "-50px";    // out of the screen
  } else {
    floaty.style.top = window.scrollTop + "px";
  }
}
```

```

}
</script>

</head>
<body>

<h1>Floaty Bar</h1>
<div id="floaty">
    This is a floaty bar
</div>

<!-- Document goes here -->
</body>
</html>

```

Cascading menus

A cascading menu should be used for large toolbars, typically for touch devices. It can also be used in cursor-based browsers, but remember that in these browsers it may take the user a while to get to the desired zone of the screen using the navigation keys.

As these menus will typically be used on touch devices, we should not use `mouseover` events to open and close the menu bar, and it is best to use `onClick` for both the opening and closing actions. We can also hide the menu when the user selects an option, scrolls the page, or moves the focus to another object.

A simple `div` show/hide interaction with JavaScript will work, or (with care) you can use a JavaScript library for this purpose.

Autocompletion

An autocomplete (or autosuggest) feature to reduce the user's typing, like the one in Figure 12-7, is a great feature, but it is not as simple to implement in a mobile site as it is in a desktop site. There are two kinds of autocompletes: preloaded and Ajax-based. The preloaded ones involve downloading all the possible values to suggest (not recommended for more than 2,000 values) and storing them in JavaScript variables, and then, if offline storage is available, storing them in the device for future usage.

The first problem is the issue of network latency and bandwidth consumption. If the user is using 2G technologies (GPRS, EDGE), the process of going to the server or preloading all the possible values can be long and costly. The second problem is the UI design, for a few reasons: generating a floating `div` over other elements can be problematic in many mobile browsers, and browsing between suggestions in nontouch devices can be difficult.



If you are going to use a JavaScript-based autocomplete solution, remember to deactivate the browser's standard autocomplete feature using `autocomplete="off"` in the text input.

All that aside, if we can save the users a lot of typing, we will be their heroes.

So, the first conclusion is that this solution is recommended only for touch-enabled smartphones, which we suppose are connected using WiFi or a 3G network. Next, we need to think about the design. The recommendation is not to use a floating `div` over other content, and instead to use a hidden `div` that replaces or pushes down the previous content. This `div` will appear just below the text box, and there must be a close button at the top-right corner.

Another thing to keep in mind for touch devices without QWERTY keyboards is that when the user has the focus in the text box (and our autocomplete feature is working), the virtual keyboard will be on the screen and there will not be much space available. One solution that will ensure that as much space as possible is available for the suggestion list is to scroll the document to the text box position when the user focuses in the text box. This will leave the text input just at the top of the screen, and with the keyboard at the bottom the middle will be open for our suggestion list (as we can see in Figure 12-7).



Figure 12-7. Google.com autocomplete on an Android device.