# 3 File editing and management: rich formatting, file storage, drag and drop

***This chapter covers***

- Rich-text HTML editing
- Location awareness with geolocation
- Working with files in a local filesystem
- Implementing drag and drop

The web is no longer merely a set of interconnected documents that people use to find information; it's also an application platform that allows developers to build web apps that anyone with a computer and browser can use. In HTML5, new standardized JavaScript APIs enable web apps to present an application interface similar to current desktop apps. Features such as rich-text editing, drag/drop interactions, local file management, and geolocation are now possible.

This chapter teaches you how to use all of these new features and APIs by walking you through the build of the Super HTML5 Editor, an HTML editor application that runs entirely on the client side, with no server-side requirements. The application allows users to manipulate HTML documents using one of two editor modes:

- A visual WYSIWYG editor for formatting text, inserting hyperlinks, adding images, and inserting maps
- An HTML markup editor for changing, adding, and deleting markup elements, useful when you need formatting or a layout feature not supported in the visual editor

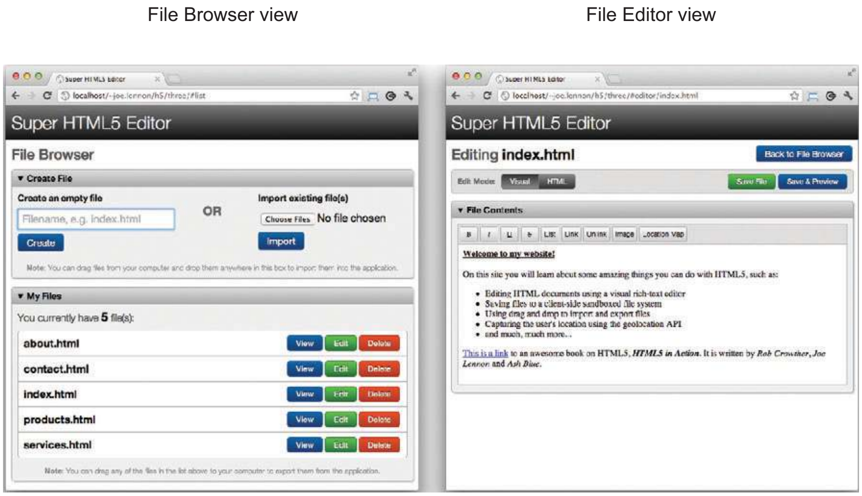
> **Why build the Super HTML5 Editor?**
>
> While working through this chapter's sample application, you'll learn to use the following:
>
> - The HTML Editing API to allow users to edit HTML markup using rich-text controls
> - The Geolocation API to capture the user's current location for use in a map
> - The File System API to provide a client-side sandbox to store the user's files
> - Drag and drop to simplify the importation and exportation of files

To make things more fun, the application also offers a client-side sandboxed filesystem where the user can create, import, export, edit, view, and delete files. To put icing on the cake, users will also be able to import and export files using drag and drop.

Let's jump right in with a high-level overview of the sample application you're going to build, followed by work on prerequisites and first steps.

## 3.1 The Super HTML5 Editor: application overview, prerequisites, and first steps

As you can see in figure 3.1, the final application will be split into two major views, the File Browser view and the File Editor view.



Note: You can drag files from your computer and drop them anywhere in this box to import them into the application.

You currently have **5** file(s):

- about.html
- contact.html
- index.html
- products.html
- services.html

Note: You can drag any of the files in the list above to your computer to export them from the application.

Editing **index.html**

**Welcome to my website!**

On this site you will learn about some amazing things you can do with HTML5, such as:

- Editing HTML documents using a visual rich-text editor
- Saving files to a client-side sandboxed file system
- Using drag and drop to import and export files
- Capturing the user's location using the geolocation API
- and much, much more...

This is a link to an awesome book on HTML5, ***HTML5 in Action***. It is written by ***Rob Crowther***, ***Joe Lennon*** and ***Ash Dixe***.

**Figure 3.1 The two views of the Super HTML5 Editor application are shown. The File Browser view (left) allows users to manipulate the files stored in the app; the File Editor view (right) enables the file to be modified using rich-text editing controls or directly using HTML markup.**

The File Browser view allows users to create empty files, import files from their computers, view a list of existing files, and perform an action on one of these files such as View, Edit, Delete, and Export. This view also provides drag-and-drop support for working with files.

The File Editor view provides two editors for manipulating the file's contents: a visual WYSIWYG editor and a raw HTML markup editor. This view also allows the user to save their changes, preview the file, and return to the File Editor view. It will also warn the user if they try to navigate away from the File Editor view when they have unsaved changes.

### Before you begin: important browser notes

The File System API (also known as the File Directories and System API) is a relatively late addition to the HTML5 specification and thus hasn't yet been implemented by most browser vendors. Although most have provided partial support for the accompanying File API, which you can use to read the contents of local files that the user selects or drops into the application, only Google Chrome currently supports the File System and File Writer APIs that are used to actually create and store files on the client side. The sample application has been written to include vendor prefixes that will probably be used when the other browsers start to include support for these features, but we can't guarantee that their actual implementation will follow this path.

Also, if you're using Chrome and plan to test this application in your local directory instead of on a server, you'll need to start Chrome with the following option:

–Allow-File-Access-From-Files

If you don't, your application's client-side filesystem will be inaccessible and the Geolocation API won't be able to access your location.

In this section, you'll build the HTML document for the application and implement basic navigation and state management functionality using JavaScript. The work happens in five steps:

- Step 1: Create index.html.
- Step 2: Add markup for the File Browser view.
- Step 3: Add markup for the File Editor view.
- Step 4: Initialize the application.
- Step 5: Enable navigation between views and manage the state of documents being edited.

### Prerequisites

Before you create the index page, you need to handle a couple of prerequisites:

1. Create a directory, and put the style.css file from this chapter's source code in it.
2. Create an empty app.js file, and put it in the same directory as the style.css file.

Note that all files for the book are available at the book's website: http://www.manning.com/crowther2.

At this stage you're probably itching to get started, so let's do just that.

### 3.1.1 Defining the HTML document structure

The initial code you need loads in the CSS and JavaScript resources for the application and defines the `<section>` elements for each of the two views.

#### STEP 1: CREATE INDEX.HTML

Begin by creating a file named index.html, and add the contents of the following listing to it.

**Listing 3.1 index.html—Application HTML structure**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Super HTML5 Editor</title>
    <link rel="stylesheet" href="style.css">
    <script src="app.js"></script>
</head>
<body class="browser-view">
    <header><h1>Super HTML5 Editor</h1></header>
    <section id="list">

    </section>
    <section id="editor">

    </section>
</body>
</html>
```

**The value of class will determine the currently displayed view. Navigating between views will be implemented later in the section.**

**The File Browser view markup should be inserted here.**

**The File Editor view markup should be inserted here.**

#### STEP 2: ADD MARKUP FOR THE FILE BROWSER VIEW

The File Browser view is split into two zones. The first zone contains two forms:

- A form for creating an empty file
- A form for importing a file from the user's computer

The second zone includes a list of files that the user has created or imported. To build these zones you'll use the `<details>` and `<summary>` *elements*, both of which are new in HTML5. The `<details>` element allows you to create a collapsible section in your code, which would previously have only been possible using a combination of JavaScript and CSS. Adding a `<summary>` element within `<details>` will put a label on the expanded `<details>` content. Add the code from the next listing to the index.html file, inside the `<section>` element with the ID attribute value `list`.

**Listing 3.2 index.html–File Browser view markup**

```
<h1>File Browser</h1>
<details open id="filedrop">
    <summary>Create File</summary>
    <form name="create">
        <div>
```

**This zone will be a target drop zone for files later in this chapter. The open attribute on the <details> element sets it to be expanded by default.**

**The create form allows users to create a new empty file.**

```
            <h2>Create an empty file</h2>
            <input type="text" name="name" placeholder=" e.g. index.html">
            <input type="submit" value="Create">
        </div>
    </form>
    <div class="spacer">OR</div>
    <form name="import">
        <div>
            <h2>Import existing file(s)</h2>
            <input type="file" name="files" multiple accept="text/html">
            <input type="submit" value="Import">
        </div>
    </form>

    <div class="note">
        <strong>Note</strong>: You can drag files from your computer and
        drop them anywhere in this box to import them into the application.
    </div>
</details>

<details open>
    <summary>My Files</summary>
    <div class="note top">
        You currently have <span id="file_count">0</span> file(s):
    </div>
    <ul id="files"></ul>
    <div class="note">
        <strong>Note</strong>: You can drag any of the files in the list
        above to your computer to export them from the application.
    </div>
</details>
```

The import form allows users to import files from their computer.

This will be populated later with a list of files.

### Step 3: Add markup for the File Editor view

This File Editor view features a switch button that allows the user to change between Visual edit mode and HTML edit mode. In Visual mode, the editor will behave much like a basic word processor, and it includes buttons for formatting the content in bold, italic, and so forth. Each button has an attribute named `data-command`, which is an example of an HTML5 data attribute. These attributes make it easy to associate primitive data with an HTML element, and an accompanying JavaScript API makes it a breeze to get back this data when it's needed. The code for the File Editor view is shown in the following listing and should be added to index.html, inside the `<section>` element with the ID attribute value `editor`.

**Listing 3.3 index.html–File Editor view markup**

```
<h1>Editing <span id="file_name"></span><a href="#list">Back to File
     Browser</a></h1>
<div class="mode-toolbar">
    <div class="left">
        <div>Edit Mode:</div>
        <button id="edit_visual" class="split_left active">Visual</button>
        <button id="edit_html" class="split_right">HTML</button>
    </div>
```

Two buttons allow the user to switch between Visual and HTML edit modes.

```
    <div class="right">
        <button id="file_save" class="green">Save File</button>
        <button id="file_preview">Save &amp; Preview</button>
    </div>
</div>

<details open>
    <summary>File Contents</summary>
    <div id="file_contents">
        <div id="file_contents_visual">
            <div id="file_contents_visual_toolbar">
                <button data-command="bold"><strong>B</strong></button>
                <button data-command="italic"><em>I</em></button>
                <button data-command="underline"><u>U</u></button>
                <button data-command="strikethrough"><del>S</del></button>
                <button data-command="insertUnorderedList">List</button>
                <button data-command="createLink">Link</button>
                <button data-command="unlink">Unlink</button>
                <button data-command="insertImage">Image</button>
                <button data-command="insertMap">Location Map</button>
            </div>
            <iframe id="file_contents_visual_editor"></iframe>
        </div>
        <div id="file_contents_html">
            <textarea id="file_contents_html_editor"></textarea>
        </div>
    </div>
</details>
```

**Contains several buttons that allow the user to format the currently selected content in the editor window.**

**The visual editor is an <iframe> element, which will later be made editable using the designMode property.**

**The HTML markup editor is a regular <textarea> element.**

With the two views defined, you can now implement JavaScript code to enable navigation between them.

## 3.1.2 Implementing navigation and state management in JavaScript

First, let's create an anonymous function block to ensure that the application doesn't pollute the global JavaScript namespace. This block will initialize the application when the DOM has finished loading.

### STEP 4: INITIALIZE THE APPLICATION

Create a new file named app.js and save it in the same directory as the index.html file you created previously. Add the contents of the following listing to this file.

**Listing 3.4 app.js–Application initialization code**

```
(function() {
    var SuperEditor = function() {

    };

    var init = function() {
        new SuperEditor();
    }

    window.addEventListener('load', init, false);
})();
```

**This constructor function is where the rest of the app's code should be inserted.**

### Step 5: Enable navigation between views, manage the state of documents being edited

With the code to initialize the application out of the way, let's add code to keep track of whether the user has made changes to a document and to switch between the File Browser and File Editor views. The code in the next listing should be added inside the SuperEditor constructor function that you created in the previous listing.

**Listing 3.5 app.js—View navigation and state management code**

```
var view, fileName, isDirty = false,
    unsavedMsg = 'Unsaved changes will be lost. Are you sure?',
    unsavedTitle = 'Discard changes';

var markDirty = function() {
    isDirty = true;
};

var markClean = function() {
    isDirty = false;
};

var checkDirty = function() {
    if(isDirty) { return unsavedMsg; }
};

window.addEventListener('beforeunload', checkDirty, false);

var jump = function(e) {
    var hash = location.hash;

    if(hash.indexOf('/') > -1) {
        var parts = hash.split('/'),
            fileNameEl = document.getElementById('file_name');

        view = parts[0].substring(1) + '-view';
        fileName = parts[1];
        fileNameEl.innerHTML = fileName;
    } else {
        if(!isDirty || confirm(unsavedMsg, unsavedTitle)) {
            markClean();
            view = 'browser-view';
            if(hash != '#list') {
                location.hash = '#list';
            }
        } else {
            location.href = e.oldURL;
        }
    }

    document.body.className = view;
};

jump();

window.addEventListener('hashchange', jump, false);
```

These variables will store the current view and filename (if in the File Editor view) and a marker to indicate if the document has been modified (isDirty).

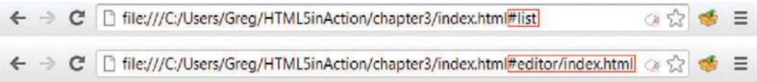
If the user tries to close the window or navigate to another page, you'll check to see if they've made unsaved changes and warn them first if necessary.

The jump event handler uses hashes in the URL to switch between the two views.

If the URL hash contains a forward slash, it should show the File Editor view for the file after the slash (if it exists).

Use the class attribute on the <body> element to indicate which is the current view–the CSS will take care of showing/hiding the views as necessary.

The jump function is called on page load and whenever the URL hash changes.



file:///C:/Users/Greg/HTML5inAction/chapter3/index.html#list

file:///C:/Users/Greg/HTML5inAction/chapter3/index.html#editor/index.html

**Figure 3.2 When you load the application right now, a hash value `#list` will be appended to the end of the URL. To navigate to the editor view manually, change this to `#editor/index.html` as shown.**

### TRY IT OUT

At this point, you should be able to navigate around the application. One slight inconvenience is that you won't be able to easily get to the File Editor view just yet, because you haven't added any of the File System functionality. To cheat your way around this, modify the URL manually, changing the `#list` at the end to `#editor/index.html`, as illustrated in figure 3.2.

With a modest amount of effort, you've roughed out the basic HTML structure, navigation functions, and state management for the application. In the next section, you'll discover how to enable the visual editor and connect it to the HTML editor, how to implement the formatting buttons, and how to use geolocation to insert a map of the user's current position coordinates.

## 3.2 Rich-text editing and geolocation

The visual editor in this chapter's sample application will allow users to write and edit rich-text content using formatting buttons that are similar to those in most word-processing applications. After formatting the document, users may need to see the underlying HTML markup to make adjustments, so the application will enable them to switch between the visual editor and the HTML editor. Also, so that you can at least get your hands dirty with the Geolocation API, we'll have you add into the application a button that inserts a location map.

### In this section, you'll learn

- To use the `designMode` property to signal to the browser that an HTML document is editable
- To use the Editing API's `execCommand` method to provide rich-text editing controls
- To use the Geolocation API

The work happens in three steps:

- Step 1: Turn `designMode` on and synchronize the content of both editors.
- Step 2: Implement the rich-text editing toolbar in the visual editor.
- Step 3: Use geolocation to insert a map of the user's location.

### 3.2.1 Using designMode to make an HTML document editable

To facilitate the visual editor mode in your app, you need to allow users to directly edit the HTML document without needing to use HTML markup. In order to make this work, you need to take advantage of a JavaScript object property, `designMode`. When you set this property's value to `on` for a given document, the entire document becomes editable, including its `<!DOCTYPE>` declaration, and `<head>` section. You'll use this property with our visual editor's `<iframe>` to make the entire contents of the `<iframe>` editable.

> **NOTE** If you need to edit the contents of only a specific HTML element, then use the `contenteditable` attribute. Although `contenteditable` is new in HTML5, it started out as a proprietary extension in IE and was later adopted by other browser vendors. As a result, browser support for it is widespread, so you can use it without fear of leaving anyone behind.

Setting `designMode` to `on` is straightforward, but you also need to build logic that will connect the visual editor to the HTML markup editor so that any changes are synced across them when appropriate. You also need to implement the switch button to allow the user to switch between the two editor modes. Enough chat about what you need to do—let's go ahead and do it.

#### Step 1: Turn designMode on and synchronize the content of both editors

In the app.js file, add the following code immediately after the line `window.addEventListener('hashchange', jump, false)`.

**Listing 3.6 app.js—Enabling `designMode` and connecting the two editors**

```
var editVisualButton = document.getElementById('edit_visual'),
    visualView = document.getElementById('file_contents_visual'),
    visualEditor = document.getElementById('file_contents_visual_editor'),
    visualEditorDoc = visualEditor.contentDocument,
    editHtmlButton = document.getElementById('edit_html'),
    htmlView = document.getElementById('file_contents_html'),
    htmlEditor = document.getElementById('file_contents_html_editor');

visualEditorDoc.designMode = 'on';

visualEditorDoc.addEventListener('keyup', markDirty, false);
htmlEditor.addEventListener('keyup', markDirty, false);

var updateVisualEditor = function(content) {
    visualEditorDoc.open();
    visualEditorDoc.write(content);
    visualEditorDoc.close();
    visualEditorDoc.addEventListener('keyup', markDirty, false);
};

var updateHtmlEditor = function(content) {
    htmlEditor.value = content;
};
```

**Enable editing of the visual editor iframe by switching on its designMode property.**

**Mark the file as dirty whenever the user makes changes to either editor.**

**This function updates the visual editor content. Every execution of updateVisualEditor constructs a new document, so you must attach a new keyup event listener.**

**This function updates the HTML editor content.**

```
var toggleActiveView = function() {
    if(htmlView.style.display == 'block') {
        editVisualButton.className = 'split_left active';
        visualView.style.display = 'block';
        editHtmlButton.className = 'split_right';
        htmlView.style.display = 'none';
        updateVisualEditor(htmlEditor.value);
    } else {
        editHtmlButton.className = 'split_right active';
        htmlView.style.display = 'block';
        editVisualButton.className = 'split_left';
        visualView.style.display = 'none';

        var x = new XMLSerializer();
        var content = x.serializeToString(visualEditorDoc);
        updateHtmlEditor(content);
    }
}

editVisualButton.addEventListener('click', toggleActiveView, false);
editHtmlButton.addEventListener('click', toggleActiveView, false);
```

**This event handler toggles between the visual and HTML editors. When updating the HTML editor, the XMLSerializer object is used to retrieve the HTML content of the iframe element.**

### PROGRESS CHECK: TRY IT OUT

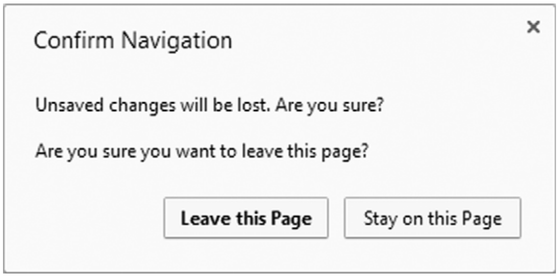
At this point, you should be able to type text in the visual editor. You'll notice that if you switch to the HTML editor, the contents should match. Similarly, if you make changes in the HTML editor and switch back to the visual editor, your changes should be shown. Try putting some arbitrary HTML styling markup in the HTML editor and notice the impact it has in the visual editor.

> **NOTE** If you try to use the formatting toolbar to style the contents of the visual editor, you'll notice that none of these buttons work. Don't fret; you'll fix that in the next section.

After you've made changes, try closing the window. You should see a warning message like the one shown in figure 3.3.

Because the saving function hasn't been implemented yet, you can ignore this warning and leave the page. You'll add the saving function in a later section.

Now that you have the basic visual and HTML editors working, let's move on and add some formatting functions to those do-nothing toolbar buttons.



Confirm Navigation

Unsaved changes will be lost. Are you sure?

Are you sure you want to leave this page?

**Figure 3.3 The `isDirty` variable we created earlier allows the application to keep track of whether the user has made changes to the document. If they've made changes and try to close the window without saving, they'll be shown this warning message to confirm they want to leave the page.**

### 3.2.2 *Providing rich-text editing controls with execCommand*

Core API

As you've already seen, the contenteditable attribute and designMode property allow developers to make any HTML element editable by the user. But up until now, all users have been able to do is type and edit text, which is hardly exciting; they've been able to do that with HTML form elements for ages! It'd be much more impressive if users could format the text using rich-text editing controls, as they would in a word processing application. That's where the Editing API method execCommand comes in.

| | | | | | |
|---|---|---|---|---|---|
| **Editing API** | 4.0 | 3.5 | 5.5 | 9.0 | 3.1 |

**EXECCOMMAND: FORMATTING AND EDITING ELEMENTS VIA CODE**

Invoking the execCommand method of an editable element applies a selected formatting command to the current selection or at the current caret position. This includes basic formatting like italicizing or bolding text and block changes like creating a bullet list or changing the alignment of a selection. ExecCommand can also be used to create hyperlinks and insert images. Basic editing commands like copy, cut, and paste can also be used by execCommand if the browser implements these features. Although the HTML5 standard specifies these editing commands, it doesn't require the browser to support them. For a full list of commands standardized in HTML5, see appendix B.

To initiate a formatting or editing action, you must pass one to three arguments to execCommand:

- The first argument, command, is a string. command contains the name of the editing or formatting action.
- The second argument, showUI, is a bool. showUI determines whether the user will see the default UI associated with command. (Some commands don't have a UI.)
- The third argument, value, is a string. execCommand will invoke command with value as its argument.

The number of required arguments for an execCommand depends on the command passed to the first argument. See appendix B or http://dvcs.w3.org/hg/editing/raw-file/tip/editing.html for a list of argument specifications for each formatting and editing command.

**STEP 2: IMPLEMENT THE RICH-TEXT EDITING TOOLBAR IN THE VISUAL EDITOR**

To use execCommand, the application will use a click event handler to pass the function name of a pressed toolbar button to execCommand's command argument. This function name will be retrieved from the button's data-command attribute. Add the code from the following listing to app.js, directly after the code you added in the previous section.

**Listing 3.7 app.js–Implementing the rich-text editing toolbar in the visual editor**

```
var visualEditorToolbar =
     document.getElementById('file_contents_visual_toolbar');

var richTextAction = function(e) {
    var command,
        node = (e.target.nodeName === "BUTTON") ? e.target :
        e.target.parentNode;

    if(node.dataset) {
        command = node.dataset.command;
    } else {
        command = node.getAttribute('data-command');
    }

    var doPopupCommand = function(command, promptText, promptDefault) {
        visualEditorDoc.execCommand(command, false, prompt(promptText,
        promptDefault));
    }

    if(command === 'createLink') {
        doPopupCommand(command, 'Enter link URL:', 'http://www.example.com');
    } else if(command === 'insertImage') {
        doPopupCommand(command, 'Enter image URL:',
        'http://www.example.com/image.png');
    } else {
        visualEditorDoc.execCommand(command);
    }
};

visualEditorToolbar.addEventListener('click', richTextAction, false);
```

**RichTextAction is the event handler for all buttons on the visual editor toolbar. When a user clicks a toolbar button, the event handler determines which button the user clicked.**

**The dataset object offers convenient access to the HTML5 data-* attributes. If the browser doesn't support this, the app falls back to the getAttribute method.**

**Because this app will require a customized UI, showUI will be set to false. The third argument, value, is passed a prompt method (of the Window object). It contains a string prompting the user for an input value and another string containing a default input value.**

### TRY IT OUT—AND CHALLENGE YOURSELF!

Core API

With the exception of the Location Map button, which you'll implement in the next section, you should be able to format the text in the visual editor to your heart's content using the rich-text editing toolbar. A few easy enhancements you could include here would be to provide support for more commands, to bind a keyboard event to a command (for example, Ctrl-B or Cmd-B could be mapped to bold), and to indicate the current selection state of the toolbar (for example, the Bold button should be depressed when the selected text is bold). To implement the latter, you can use the Editing API method `queryCommandState`, which is covered in more detail in appendix B.

## 3.2.3 *Mapping a user's current location with the Geolocation API*

Core API

To enable your application to insert a map based on the user's position, you'll need to use the Geolocation and Google Maps APIs. The Geolocation API provides the method `getCurrentPosition`, which will enable the application to obtain the user's geographic coordinates. The Google Maps API provides a querying function to return a static map from a set of submitted coordinates.

When Google Maps returns the selected map, your application will paste the map into the visual editor using the `execCommand`'s `insertImage` function.

| | | | | | |
|---|---|---|---|---|---|
| **Geolocation API** | 5.0 | 3.5 | 9.0 | 10.6 | 5.0 |

Before you dive in, we want you to know that although this sample application doesn't explore all of the features of geolocation, it does show you how simple it is to acquire a user's position and integrate it with a mapping service. If you're looking to build a more dynamic mapping app, you'll be glad to know that the Geolocation API can also support features like:

- Tracking user movement over set time intervals
- Obtaining the user's altitude, heading, and speed
- Limiting GPS use when battery life is a concern

To find out more about these geolocation features, see appendix B.

### Step 3: Use geolocation to insert a map of the user's location

To implement geolocation in your application, in the app.js file locate the `if` block that checks whether the command is `createLink`, `insertImage`, or something else. Add the following code before the last `else` and after the `}`.

**Listing 3.8 app.js–Using geolocation to insert a map of the user's location**

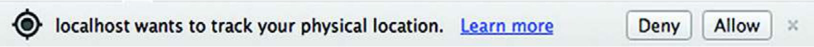
```
else if(command === 'insertMap') {
  if(navigator.geolocation) {
      node.innerHTML = 'Loading';
      navigator.geolocation.getCurrentPosition(function(pos) {
          var coords = pos.coords.latitude+','+pos.coords.longitude;
          var img = 'http://maps.googleapis.com/maps/api/staticmap?markers='
              +coords+'&zoom=11&size=200x200&sensor=false';
          visualEditorDoc.execCommand('insertImage', false, img);
          node.innerHTML = 'Location Map';
      });
  } else {
    alert('Geolocation not available', 'No geolocation data');
  }
}
```

**Check to see if the user's browser supports geolocation.**

**The getCurrentPosition method will trigger the browser to ask the user for access to the user's location. If permission is granted, getCurrentPosition executes a callback function, passing the user's location data in the form of a Position object.**

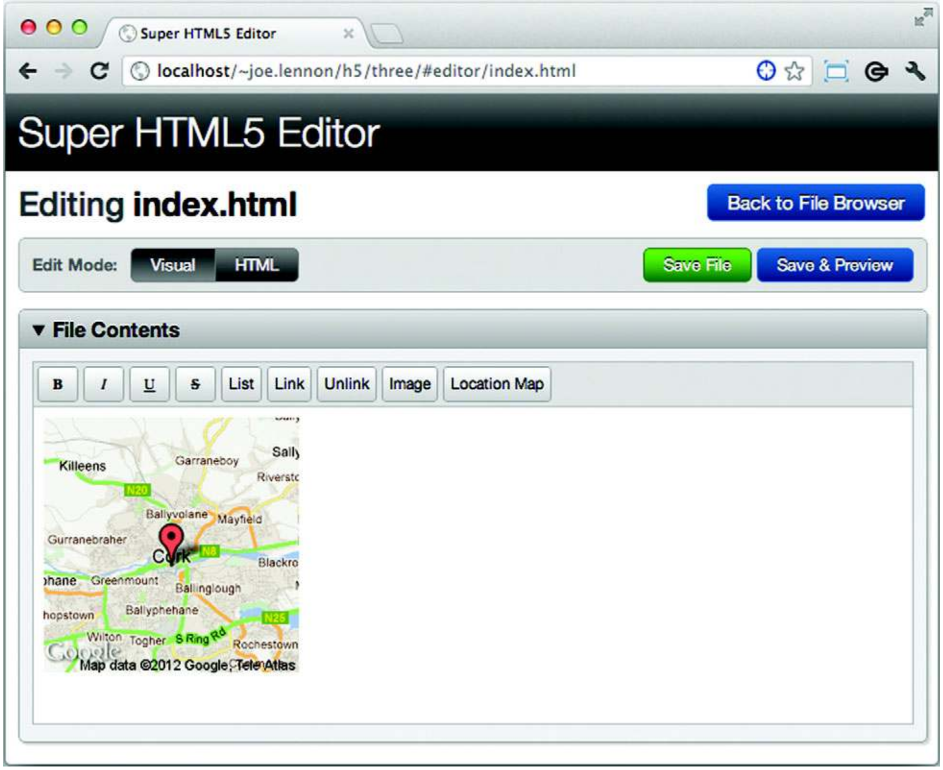
**Use execCommand to insert a static Google Maps image of the user's location.**

When the user clicks the Location Map button on the rich-text editor toolbar, the browser will request permission for the application to access their location data, as shown in figure 3.4.



localhost wants to track your physical location.

**Figure 3.4** The browser will request the user's permission to enable the Geolocation API. If access is denied, the browser will behave as though it doesn't support geolocation.



Super HTML5 Editor

localhost/~joe.lennon/h5/three/#editor/index.html

Super HTML5 Editor

Editing index.html

**Figure 3.5** A map of the user's location will be added to the editor. This map is actually an image generated by the Google Maps Static API. Easy, huh?

If the user chooses to allow access to their location, a map with a marker on their position will be added to the editor, as illustrated in the screenshot in figure 3.5. In order for the map to appear, you must click inside the editor's text box before clicking the Location Map button.

Now that users can see their location on a map and manipulate HTML documents, you need to provide a way of saving their work in actual files. In the next section, you'll learn how to use the HTML5 File System API to do just that.

## 3.3 Managing files locally: the File System, Quota Management, File, and File Writer APIs

Working with files in web applications has always been tricky. If you wanted to save a file, you'd select it using a file `<input>` element, then the browser would upload the file to the server for storage. Downloading a previously stored file was a similarly slow and cumbersome process. In addition, you were burdened with the tedious task of developing yet another file management system using one set of tools and languages on the server and another on the browser side. Suffice it to say, files and web applications have always been a bit of a bitter cocktail. Thankfully, HTML5 is going to greatly speed up this development process with the File System API.

| | | | | | |
|---|---|---|---|---|---|
| **File System API** | 13.0 | N/A | N/A | N/A | N/A |

**In this section, you'll learn**

- How to create a sandboxed filesystem using the File System API
- How to use the Quota Management API to allocate local storage space
- How to create filesystem services using the File Writer and File APIs

The File System API offers web applications access to a sandboxed storage space on the client's local filesystem. For security purposes, applications can only access the files in their own sandbox on the client, preventing malicious apps from accessing and manipulating data stored by other applications. The File System API also offers applications a choice between a temporary or persistent filesystem. Data in a temporary filesystem can be removed at any stage by the browser, and the data's continued existence shouldn't be relied on, whereas data in a persistent filesystem can only be removed if specifically requested by the user. Because we want the Super HTML5 Editor to save a user's work for later use, we'll show you how to build a persistent filesystem.

> **WARNING** The File System API was added to HTML5 much later than most APIs, and so browser support for it is far less mature. Because Chrome is the only browser currently offering any implementation of the API, the code in this section has been tested only on Chrome. Every effort has been made to ensure that it will work in other browsers at a later stage, but unfortunately we can't guarantee anything on that front.

The File System API offers almost all the needed functionality to create and manage a sandboxed filesystem except the ability to request local storage and analyze local storage availability. To do this, you need the Quota Management API.

| | | | | | |
|---|---|---|---|---|---|
| **Quota Management API** | 13.0 | N/A | N/A | N/A | N/A |

The Quota Management API enables the application to determine if enough local file storage exists to save data. If sufficient space exists, the application can use the Quota Management API to request storage via a request for quota.

> **NOTE** The File System API makes use of other file-related APIs such as the File Writer and File APIs. This section will be making calls to these underlying APIs and pointing them out as the sandboxed filesystem is built.

You'll walk through seven steps to create the filesystem:

- Step 1: Create a persistent filesystem.
- Step 2: Retrieve and display a file list.
- Step 3: Load files in the File Editor view using the File API.
- Step 4: View, edit, and delete files in the filesystem.
- Step 5: Create new, empty files in the filesystem.
- Step 6: Import existing files from the user's computer.
- Step 7: Implement the Save and Preview buttons.

### 3.3.1 Creating an application filesystem

Using the File System and Quota Management APIs, the process of creating the first part of the filesystem, the base persistent filesystem, becomes relatively straightforward and is accomplished in a single listing, listing 3.9. To help you navigate the code, look out for the following implementation process within the code:

- Assign a filesystem object to the window `fileSystem` field.
- Assign a storage and quota management object to the window `storageInfo` field.
- Set the filesystem as persistent.
- Request a quota from the local storage system.

#### STEP 1: CREATE A PERSISTENT FILESYSTEM

Core API

With the process in mind, review the following listing to see the detailed implementation. Then add the code after the call to `addEventListener('click', richTextAction, false)`.

**Listing 3.9 app.js–Creating a persistent filesystem**

```
window.requestFileSystem = window.requestFileSystem ||
     window.webkitRequestFileSystem
    || window.mozRequestFileSystem || window.msRequestFileSystem || false;
window.storageInfo = navigator.persistentStorage ||
     navigator.webkitPersistentStorage || navigator.mozPersistentStorage ||
     navigator.msPersistentStorage || false;

var stType = window.PERSISTENT || 1,
    stSize = (5*1024*1024),
    fileSystem,
    fileListEl = document.getElementById('files'),
    currentFile;
```

**For convenience, point the filesystem objects to possible vendor prefixes. If the browser doesn't support these objects, the objects will have a false value.**

**Define basic variables for use in the app: storage type and size, filesystem object, the file list element, and the currently selected file (when editing).**

```
var fsError = function(e) {
    if(e.code === 9) {
        alert('File name already exists.', 'File System Error');
    } else {
        alert('An unexpected error occured. Error code: '+e.code);
    }
};
var qmError = function(e) {
    if(e.code === 22) {
       alert('Quota exceeded.', 'Quota Management Error');
    } else {
       alert('An unexpected error occurred. Error code: '+e.code);
    }
};

if(requestFileSystem && storageInfo) {
    var checkQuota = function(currentUsage, quota) {
        if(quota === 0) {
            storageInfo.requestQuota(stType, stSize, getFS, qmError);


        } else {
            getFS(quota);
        }
    };
    storageInfo.queryUsageAndQuota(stType, checkQuota, qmError);




    var getFS = function(quota) {
        requestFileSystem(stType, quota, displayFileSystem, fsError);
    }
    var displayFileSystem = function(fs) {
        fileSystem = fs;
        updateBrowserFilesList();
        if(view === 'editor') {
            loadFile(fileName);
        }
    }
} else {
    alert('File System API not supported', 'Unsupported');
}
```

**Standard error function for all File System API method calls.**

**Standard error function for all Quota Management API method calls.**

**Check to see if the browser supports the File System API and the Quota Management API (also known as StorageInfo).**

**Because this app has a persistent filesystem, the request for quota will trigger a message asking the user's permission to access the browser's filesystem.**

**If queryUsageAndQuota successfully executes, it passes usage and quota info to the callback function, checkQuota; otherwise, qmError is called. CheckQuota determines if sufficient quota exists to store files; if not, then it needs to request a larger quota.**

**The requestFileSystem method is used to get the filesystem object.**

**You'll implement updateBrowserFilesList and displayBrowserFileList in a later section. These functions will retrieve and display files in the app's filesystem.**

**You'll implement loadFile in a later section. If the editor view is the current view, then load the file into the editor.**

Unfortunately, you aren't quite ready to test your filesystem. You need to implement some functions to retrieve and display any existing files in the app's filesystem.

### 3.3.2 Getting a list of files from the filesystem

In listing 3.9, the `displayFileSystem` function receives a reference to the filesystem object and then calls a function named `updatebrowserFilesList`. In this section, you'll create this function, which will retrieve a list of files in the app's filesystem directory and display it in the My Files zone of the File Browser.

### STEP 2: RETRIEVE AND DISPLAY A FILE LIST

You'll need the next two listings for this work: one to create the updateBrowserFilesList function, another to create the displayBrowserFileList function. First, displayBrowserFileList will accept a complete list of files as an argument and update the UI to display each of these files with View, Edit, and Delete buttons. Right after the displayFileSystem function you created previously, add the code from the next listing.

**Listing 3.10 app.js—Building the file list UI from an array of files**

```
var displayBrowserFileList = function(files) {
    fileListEl.innerHTML = '';
    document.getElementById('file_count').innerHTML = files.length;
    if(files.length > 0) {
        files.forEach(function(file, i) {
            var li = '<li id="li_'+i+'" draggable="true">'+file.name
                + '<div><button id="view_'+i+'">View</button>'
                + '<button class="green" id="edit_'+i+'">Edit</button>'
                + '<button class="red" id="del_'+i+'">Delete</button>'
                + '</div></li>';
            fileListEl.insertAdjacentHTML('beforeend', li);

            var listItem = document.getElementById('li_'+i),
                viewBtn = document.getElementById('view_'+i),
                editBtn = document.getElementById('edit_'+i),
                deleteBtn = document.getElementById('del_'+i);

            var doDrag = function(e) { dragFile(file, e); }
            var doView = function() { viewFile(file); }
            var doEdit = function() { editFile(file); }
            var doDelete = function() { deleteFile(file); }

            viewBtn.addEventListener('click', doView, false);
            editBtn.addEventListener('click', doEdit, false);
            deleteBtn.addEventListener('click', doDelete, false);
            listItem.addEventListener('dragstart', doDrag, false);
        });
    } else {
        fileListEl.innerHTML = '<li class="empty">No files to display</li>'
    }
};
```

Update the file counter with the number of files in the filesystem.

Iterate over each file in the filesystem using the forEach array function.

Draggable will be discussed in a later section on drag-and-drop interactivity.

Attach event handlers to the View, Edit, and Delete buttons and the list item itself.

Later in the chapter, you'll implement doDrag to support drag-and-drop functions.

If there are no files, show an empty list message.

Now, to execute the displayBrowserFileList function you just created, you need to pass an array of all the files in the app's directory. The updateBrowserFilesList function will do just that, using a DirectoryReader object and reading the list of files one set of files at a time until all files in the app's directory have been read. Add the code from the next listing right after the displayBrowserFileList function.

**Listing 3.11 app.js—Reading the file list using the directory reader**

```
var updateBrowserFilesList = function() {
    var dirReader = fileSystem.root.createReader(),
        files = [];
```

Create a directory reader. Later in the listing, you'll use it to get the complete list of files.

The directory listing is read in one set of files at a time, so you'll use a recursive function to keep reading until all files have been retrieved.

```
    var readFileList = function() {
        dirReader.readEntries(function(fileSet) {
            if(!fileSet.length) {
                displayBrowserFileList(files.sort());
            } else {
                for(var i=0,len=fileSet.length; i<len; i++) {
                    files.push(fileSet[i]);
                }
                readFileList();
            }
        }, fsError);
    }
    readFileList();
};
```

When the end of the directory is reached, call the displayBrowserFileList function, passing the alphabetically sorted files array as an argument.

If you're not at the end of the directory, push the files just read into the files array and recursively call the readFileList function again.

Next, you'll discover how to implement the View, Edit, and Delete buttons displayed for each of the files in the filesystem.

### 3.3.3 Loading, viewing, editing, and deleting files

Back in the `displayFileSystem` function in listing 3.9, you may have noticed an `if` block that called a function named `loadFile` if the current view was the editor view. Let's go ahead and implement that function now, as well as some small functions that will allow users to view, edit, and delete files in the filesystem.

#### STEP 3: LOAD FILES IN THE FILE EDITOR VIEW USING THE FILE API

Core API

The `loadFile` function uses the File System API method `getFile` to retrieve the FileEntry from the filesystem. In order to read the file contents, `loadFile` uses the File API method `readAsText`. Lastly, `loadFile` displays the file contents to the visual and HTML editors. Add the code from the following listing to app.js right after the `updateBrowserFilesList` function you added previously.

| | | | | | |
|---|---|---|---|---|---|
| **File API** | 13.0 | 3.6 | N/A | 11.1 | N/A |

**Listing 3.12 app.js—Loading files in the File Editor view**

The getFile method takes four arguments: (1) relative or absolute path to filename, (2) options object ({create: boolean, exclusive: boolean}—both default to false), (3) success callback function, and (4) error callback function. If a FileEntry is found, getFile passes the selected FileEntry to the fileEntry argument of the success callback function. See table 3.1 for a list of possible options arguments and their effect on getFile behavior.

```
var loadFile = function(name) {
    fileSystem.root.getFile(name, {}, function(fileEntry) {
        currentFile = fileEntry;
        fileEntry.file(function(file) {
            var reader = new FileReader();
            reader.onloadend = function(e) {
                updateVisualEditor(this.result);
```

The file method of the File System API is used to retrieve the file from the fileEntry and pass the file to the callback function.

A FileReader object, reader, is used to read the contents of the file. When reader is done, it triggers the onloadend event handler to update the visual and HTML editors.

```
                updateHtmlEditor(this.result);
            }
            reader.readAsText(file);
        }, fsError);
    }, fsError);
};
```

**With a new FileReader created and its onloadend event defined, call readAsText to read the file and load it into reader's `result` attribute.**

Table 3.1 reviews the behavior of the File System API method `getFile` when passed different values of the options object. The object consists of two Boolean fields. The first, `create`, determines if `getFile` should try to create a new FileEntry object (`create:true`) or retrieve an existing FileEntry object (`create:false`). The second field, `exclusive`, determines if `getFile` should check for the existence of a FileEntry object with the same file path name as `getFile`'s filename argument (`exclusive:true`).

**Table 3.1 A list of `getFile`'s responses to various configurations of the `options` argument**[a]

| FileEntry state | options object | getFile response |
|---|---|---|
| FileEntry found at given file path name | create: false<br>exclusive ignored | FileEntry is returned |
| | create: true<br>exclusive: true | Error is thrown |
| FileEntry found at given file path name, but the FileEntry is a directory | create: false<br>exclusive ignored | Error is thrown |
| No FileEntry found at given file path name | create: false<br>exclusive ignored | Error is thrown |
| | create: true<br>exclusive ignored | FileEntry created exclusive ignored and returned[b] |

[a] http://www.w3.org/TR/file-system-api/ .

[b] You cannot create a FileEntry if its immediate parent directory doesn't exist.

We know that at this point you may be thinking, "When am I going to be able to test this code?" Just a few more sections, we promise.

### STEP 4: VIEW, EDIT, AND DELETE FILES IN THE FILESYSTEM

Core API

The code to view, edit, and delete files in the filesystem is quite straightforward. The three functions in listing 3.13 use two File System API methods: `toURL` and `remove`.

- The `toURL` method retrieves a URL location at which the file resource can be accessed. Using `toURL` is really convenient for viewing files. It saves you from having to read the contents of the file and display it using JavaScript. Instead, you can invoke a popup window and pass the URL location to it.
- The `remove` method deletes the file and executes a callback when it's done.

To implement the view, edit, and delete functionality, add the code from the next listing to app.js right after the `loadFile` function.

**Listing 3.13 app.js—Viewing, editing, and deleting files**

```
var viewFile = function(file) {
    window.open(file.toURL(), 'SuperEditorPreview', 'width=800,height=600');
};

var editFile = function(file) {
    loadFile(file.name);
    location.href = '#editor/'+file.name;
};

var deleteFile = function(file) {
    var deleteSuccess = function() {
        alert('File '+file.name+' deleted successfully', 'File deleted');
        updateBrowserFilesList();
    }

    if(confirm('File will be deleted. Are you sure?', 'Confirm delete')) {
        file.remove(deleteSuccess, fsError);
    }
};
```

**The toURL method makes it a breeze to view the contents of a file, because you can simply launch it in a new browser window.**

**To edit the file, you load the file into the visual and HTML editors and make the File Editor view active by changing the URL hash.**

**When the remove function has completed, it will execute the deleteSuccess callback function, which calls the updateBrowserFilesList function to ensure the listing is updated.**

If you've been trying to test this functionality as you made your way through the section, you may have found it difficult given that there are no files to load, view, edit, or delete! Next, you'll learn how to create new empty files and how to allow users to import existing files from their computer using a traditional file `<input>` element.

## 3.3.4 Creating new files

There are two ways of creating new files in the File System API. The first is to create a new, empty file. The second is to allow the user to import an existing file from their computer using a file `<input>` element. You'll now implement both of these options, starting with creating empty files.

### STEP 5: CREATE NEW, EMPTY FILES IN THE FILESYSTEM

In listing 3.12 you saw how the `getFile` method returns a FileEntry object for a given filename if it exists:

```
var loadFile = function(name) {
    fileSystem.root.getFile(name, {}, function(fileEntry) {...
```

You can also use `getFile` to create a new FileEntry, if it doesn't exist, by passing a configuration object to the method. The code in listing 3.14 shows how to do this. The logic for creating a new file will be placed in the event handler, `createFormSubmit`, and attached to the File Browser create button. `CreateFormSubmit` will perform basic validation to ensure that the user is creating an HTML file and that the file doesn't already exist, and if all validation passes, it will create the file. Add this code directly after the `deleteFile` function.

**Listing 3.14 app.js—Creating a new empty file**

```
var createFile = function(field) {
    var config = {
        create: true,
        exclusive: true
    };

    var createSuccess = function(file) {
        alert('File '+file.name+' created successfully', 'File created');
        updateBrowserFilesList();
        field.value = '';
    };

    fileSystem.root.getFile(field.value, config, createSuccess, fsError);
};

var createFormSubmit = function(e) {
    e.preventDefault();
    var name = document.forms.create.name;
    if(name.value.length > 0) {
        var len = name.value.length;
        if(name.value.substring(len-5, len) === '.html') {
            createFile(name);
        } else {
            alert('Only extension .html allowed', 'Create Error');
        }
    } else {
        alert('You must enter a file name', 'Create Error');
    }
};

document.forms.create.addEventListener('submit', createFormSubmit, false);
```

The config object is passed to the getFile method, telling getFile to create a FileEntry, but only if a FileEntry with that name doesn't exist.

When the getFile method returns successfully, display a confirmation message, reload and display the files list, and clear the form field.

This is the event handler for the File Browser create button. When the create form is submitted, perform validation, and if it passes, call the createFile function.

### PROGRESS CHECK: TRY IT OUT!

Finally! You can test the code! You should be able to create empty files using the form on the File Browser view, as illustrated in figure 3.6. When the file has been created, you should be able to view it (it will be just an empty document, of course), edit it (although you won't be able to save changes just yet), and delete it.

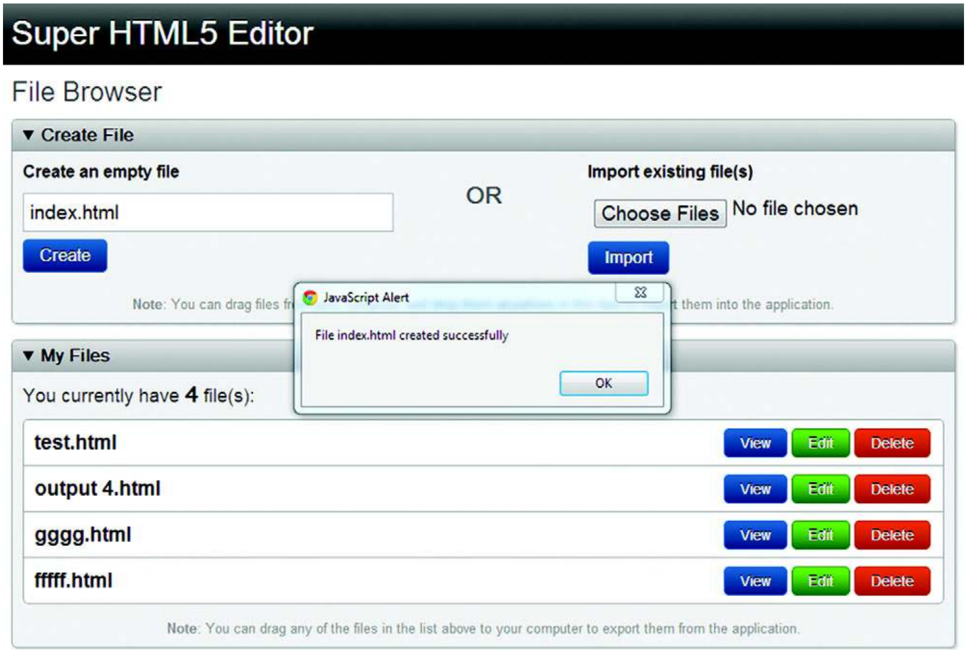
The app is finally starting to take shape! Next, let's see how you can allow a user to import existing files on their computer into the application.

### STEP 6: IMPORT EXISTING FILES FROM THE USER'S COMPUTER

Core API

Importing files from the user's computer is a little more complicated than creating an empty file. You need to create a FileEntry and then write the contents of the imported file to the FileEntry using the File Writer API.

| | | | | | |
|---|---|---|---|---|---|
| **File Writer API** | 13.0 | N/A | N/A | N/A | N/A |



Super HTML5 Editor

File Browser

▼ Create File

Create an empty file

index.html

OR

Import existing file(s)

No file chosen

Note: You can drag files fr... t them into the application.

JavaScript Alert

File index.html created successfully

▼ My Files

You currently have **4** file(s):

test.html

output 4.html

gggg.html

fffff.html

Note: You can drag any of the files in the list above to your computer to export them from the application.

**Figure 3.6 The file index.html has been successfully created!**

In addition, because you added the `multiple` attribute to the File Browser Import form,

```
...<form name="import">
        <div>
            <h2>Import existing file(s)</h2>
            <input type="file" name="files" multiple accept="text/html">
            <input type="submit" value="Import">
        </div>...
```

you must handle the possibility of importing multiple files at one time. Although implementing this isn't difficult, the validation process becomes more complicated, as you'll see. Copy the following code, and insert it right after the event listener you added to the create form in the previous section.

**Listing 3.15 app.js—Importing files from the user's computer**

```
var importFiles = function(files) {
    var count = 0, validCount = 0;

    var checkCount = function() {
        count++;
        if(count === files.length) {
            var errorCount = count - validCount;
            alert(validCount+' file(s) imported. '+errorCount+'
            error(s) encountered.', 'Import complete');
            updateBrowserFilesList();
        }
    };
```

If all of the files have been checked, show how many were imported and how many failed and update the file list.

```
    for(var i=0,len=files.length;i<len;i++) {
        var file = files[i];

        (function(f) {
            var config = {create: true, exclusive: true};
            if(f.type == 'text/html') {
                fileSystem.root.getFile(f.name, config,
                function(theFileEntry) {
                    theFileEntry.createWriter(function(fw) {
                        fw.write(f);
                        validCount++;
                        checkCount();
                    }, function(e) {
                        checkCount();
                    });
                }, function(e) {
                    checkCount();
                });
            } else {
                checkCount();
            }
        })(file);
    }
};
```

Loop through the files the user has selected and attempt to create them in the app's filesystem.

Because this for loop may execute a callback function that uses a file object, f, defined by the loop, and because an iteration of the loop may finish before the callback has fired, a closure was implemented to preserve the file object state.

GetFile creates a new FileEntry in the app's filesystem, and then createWriter creates a FileWriter for the FileEntry. At this point, you can copy the imported file, f, by calling the FileWriter method, write, and passing f as an argument.

```
var importFormSubmit = function(e) {
    e.preventDefault();
    var files = document.forms.import.files.files;
    if(files.length > 0) {
        importFiles(files);
    } else {
        alert('No file(s) selected', 'Import Error');
    }
};

document.forms.import.addEventListener('submit', importFormSubmit, false);
```

Read the files from the file's <input> element and call the importFiles function if at least one file has been selected.

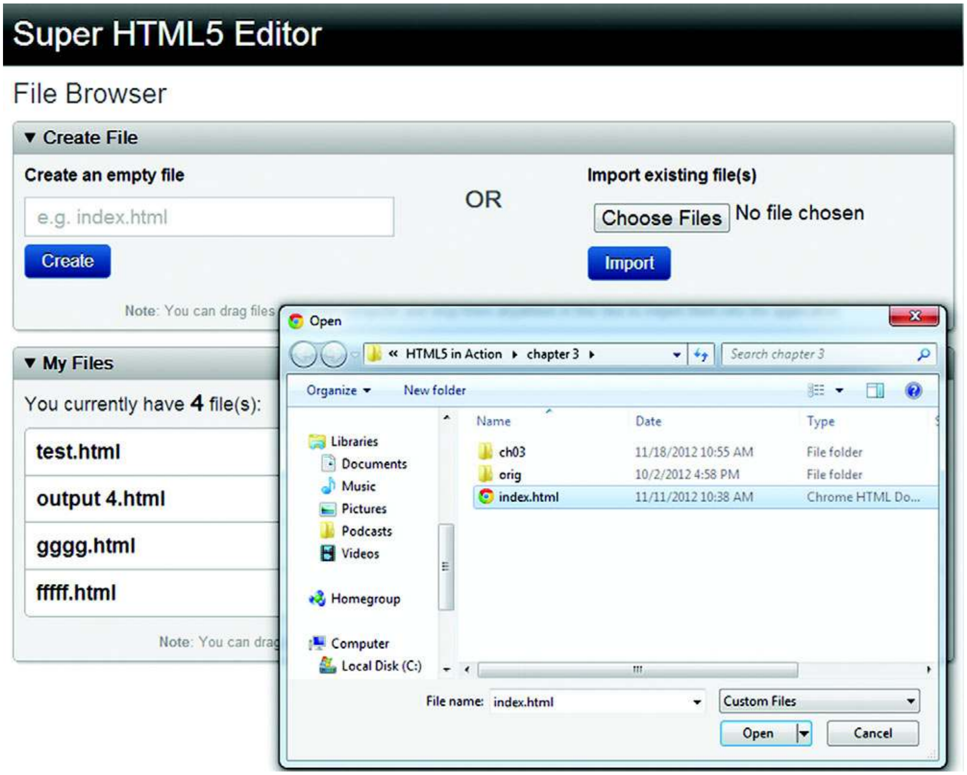
At this point you should be able to import existing HTML files from your computer into the application. You should also be able to view, edit (well, you can view in the File Editor view; you won't be able to save changes just yet), and delete files. Figure 3.7 illustrates the dialog window that pops up when you click the Choose Files button.

### 3.3.5 Saving files using the File Writer API

The final part of the filesystem functionality you need to add to the application is saving files in the File Editor view using the File Writer API. You've already seen the File Writer API in action; in the previous section when importing files from the user's computer, you used the File Writer API to save the contents of existing files into the newly created files in the application's filesystem. Now you'll use a similar approach to implement the Save and Preview buttons in the File Editor view of the application.

#### STEP 7: IMPLEMENT THE SAVE AND PREVIEW BUTTONS

To implement the Save and Preview buttons, add the code from the next listing just after the event listener you added to the import form in the previous section.



Super HTML5 Editor

File Browser

▼ Create File

Create an empty file

e.g. index.html

OR

Import existing file(s)

No file chosen

Note: You can drag files

▼ My Files

You currently have **4** file(s):

test.html

output 4.html

gggg.html

fffff.html

Note: You can drag

Open

« HTML5 in Action ▸ chapter 3 ▸

| Name | Date | Type |
|---|---|---|
| ch03 | 11/18/2012 10:55 AM | File folder |
| orig | 10/2/2012 4:58 PM | File folder |
| index.html | 11/11/2012 10:38 AM | Chrome HTML Do... |

File name: index.html

Custom Files

**Figure 3.7 After the user clicks the Choose Files button, a dialog window pops up.**

**Listing 3.16 app.js—Saving files using the File Writer API**

```
var saveFile = function(callback) {
    var currentView = function() {
        if(htmlView.style.display === 'block') {
            return 'html';
        } else {
            return 'editor';
        }
    }

    var content;

    if(currentView() === 'editor') {
        var x = new XMLSerializer();
        content = x.serializeToString(visualEditorDoc);
    } else {
        content = htmlEditor.value;
    }

    currentFile.createWriter(function(fw) {
        fw.onwriteend = function(e) {
```

**Check if the currently displayed view is the visual or HTML editor.**

**Get the contents of the relevant editor.**

**When the file writer, fw, finishes resetting the file's length to zero, fw triggers the onwriteend event handler. This event handler redefines fw's onwriteend event handler and then saves the file by calling write.**

```
            fw.onwriteend = function(e) {
                if(typeof callback === 'function') {
                    callback(currentFile);
                } else {
                    alert('File saved successfully', 'File saved');
                }
                isDirty = false;
               };
                  var blob = new Blob([content],
                                     {text: 'text/html', endings:'native'});
            fw.write(blob);
       };
       fw.onerror = fsError;
       fw.truncate(0);
    }, fsError);
};

var previewFile = function() {
    saveFile(viewFile);
};

var saveBtn = document.getElementById('file_save');
var previewBtn = document.getElementById('file_preview');

saveBtn.addEventListener('click', saveFile, false);
previewBtn.addEventListener('click', previewFile, false);
```

**When file writer, fw, has finished writing content to the currentfile, fw triggers the event handler for onwriteend. Callback refers to the callback function passed to the saveFile function.**

**Use a Blob to construct a blob object from content, a string-based representation of the editor's content.**

**Use the endings parameter to specify what type of end-of-line marker should be used. A value of native instructs a Blob constructor to use an end-of-line marker native to the browser's underlying OS.**

**Before saving data with file writer, fw, use truncate(0) to ensure its length attribute is set to zero. Otherwise, when the application saves a file that's shorter than its previous version, the length attribute will be unchanged. As a result, you'd see old text filling in the gap between the new shorter file and its previous longer version.**

**SaveFile has been passed a callback function, viewFile. It's called when saveFile has finished writing the editor contents to currentFile.**

The filesystem functionality of the application is now complete. You should be able to create, load, view, edit, save, and delete HTML files using the app. If you want to take the application further, you could easily extend it so that it supports multiple directories, allows editing of additional file types (CSS and JavaScript support would be nice), and provides syntax highlighting of the HTML markup. There are a plethora of opportunities for expansion.

We'll wrap up this chapter in the next section by adding a jazzy extra—drag-and-drop support.

## 3.4 Adding drag-and-drop interactivity

Drag-and-drop interactions are a popular feature in computer applications. For example, consider the GUIs of current OSes. They allow you to move files, documents, and applications around by dragging them from one location and dropping them to another. In Mac OS X, if you have an external hard drive plugged into your computer, you can eject it by dragging it to the trash icon in the dock.

In recent years, web applications have started to provide drag-and-drop support. Common examples are copying/moving items from one list to another; rearranging the order of a list; moving regions of the page around for a customized experience; and moving images, files, or documents to virtual directories in content management systems. Up until now, developers had to rely on using JavaScript frameworks to provide web apps with decent drag-and-drop features. In HTML5, however, a full Drag and Drop API has been specified to supplant these JavaScript frameworks.

| | | | | | |
|---|---|---|---|---|---|
| **Drag and Drop API** | 4.0 | 3.5 | 5.5 | 12.0 | 3.1 |

In this section, you'll use the Drag and Drop API to enhance the Super HTML5 Editor application by

- Enabling users to import files into the application by dragging them in from their computer
- Allowing users to export files from the application by dragging them to their computer

### 3.4.1 Dragging files into an application for import

Core API

To allow users to drag files into the application, you need to create a target zone or drop zone where the user can drag the files and expect them to be imported. If you've already loaded the application in your browser, you'll probably have noticed a note at the bottom of the Create File zone in the File Browser view. The note informs users to import files by dropping them anywhere in this zone. Let's stay true to our word and provide this functionality.

To enable the Create File zone, you need to implement two event handlers for the zone: one for the `drop` event and another for the `dragover` event. The `drop` event handler will enable the application to import files that are dropped into the Create File zone, and the `dragover` event handler will signal a pending copy operation to the app. The app will respond to the signal by adding a copy decal to the file icon(s) being dragged into the Create File zone.

Add the code in the following listing right after the line `previewBtn.addEventListener('click', previewFile, false)`.

**Listing 3.17 app.js—Allowing users to import files by dropping them in the application**

```
var fileDropZone = document.getElementById('filedrop');

var importByDrop = function(e) {

    e.stopPropagation();
    e.preventDefault();

    var files = e.dataTransfer.files;

    if(files.length > 0) {
        importFiles(files);
    }
};
```

**Designate the drop zone for files as the element with the ID filedrop.**

**When files are dropped into the browser window, the default browser behavior is to load the files and navigate away from the app, so you need to cancel this default behavior. First, invoke stopPropagation to prevent the drop event from bubbling up to any ancestor elements of fileDropZone. Second, invoke preventDefault to stop the browser from calling the default event handler attached to fileDropZone.**

**If the user is dragging files, these will reside in the dataTransfer object. To load them into the app, pass them to the importFiles function (defined in listing 3.15).**

```
var importDragOver = function(e) {
    e.preventDefault();
    e.dataTransfer.effectAllowed = 'copy';

    e.dataTransfer.dropEffect = 'copy';
    return false;
};
```

**Because you want the imported file(s) to be copied when they're dropped into the zone, set the dragover event properties, effectAllowed and dropEffect, to copy. When the user drags the file over the drop zone, the file image(s) will change to indicate a pending copy operation.**

```
fileDropZone.addEventListener('drop', importByDrop, false);
fileDropZone.addEventListener('dragover', importDragOver, false);
```

**TRY IT OUT!**

With this code added to your app, try it out by dragging an HTML file from your computer into the designated drop zone. If a file with the same name doesn't exist, it should be successfully imported into the filesystem, just as if you had manually selected the file using the regular file `<input>` dialog box. You can even drag multiple files into the application at a time. Next, you'll wrap things up by enabling users to export files by dragging them out of the application.

### 3.4.2 Dragging files out of an application for export

Core API

Some of the groundwork for your export drag-and-drop functionality has already been set. In listing 3.10 in the `displayBrowserFileList` function, you added code that created a new list item for each of the files in the filesystem. If you look at this code, you'll notice that the `<li>` element you constructed has an attribute, `draggable`, set to `true`:

```
...
files.forEach(function(file, i) {
    var li = '<li id="li_'+i+'" draggable="true">'+file.name
        + '<div><button id="view_'+i+'">View</button>'
        + '<button class="green" id="edit_'+i+'">Edit</button>'
        + '<button class="red" id="del_'+i+'">Delete</button>'
        + '</div></li>';
...
```

In addition, you'll see that a listener was added to the `dragstart` event of this item:

```
...
var doDrag = function(e) { dragFile(file, e); }
...
listItem.addEventListener('dragstart', doDrag, false);...
```

Believe it or not, all you need to do to implement the export functionality is to define the `dragFile` function. One last time, add the code in the next listing to app.js, right after the line `fileDropZone.addEventListener('dragover', importDragOver, false)`.

**Listing 3.18 app.js—Allowing users to export files by dragging them out of the app**

```
var dragFile = function(file, e) {
    e.dataTransfer.effectAllowed = 'copy';
    e.dataTransfer.dropEffect = 'copy';
    e.dataTransfer.setData('DownloadURL', 'application/octet-
     stream:'+file.name+':'+file.toURL());
};
```

When the user starts dragging a draggable item in the app, the setData method of the dataTransfer object can be used to define what data should be dropped.

If you were hoping for more code than that to implement the export functionality, you're probably disappointed—that really is all you need. The `toURL` method that was used previously in the `viewFile` method is put to use again, this time to construct a downloadable object (`DownloadURL`) that's saved to the user's computer. Be sure to give it a try; drag one of the files out of your application and drop it on your computer's desktop.

At long last the application is complete. At this point you should have a fully functional web-based HTML editor that allows you to import and export files using drag and drop.

## 3.5 Summary

Not long ago the idea that you could build a full client-side WYSIWYG HTML editor application featuring the ability to create, edit, save, and drag/drop files was nothing more than a daydream for web application developers. In HTML5 this is all now a reality, and as browser support steadily improves, we're getting closer to a situation where users will come to expect features like these to be a part of every web application. Progressive functionality like this will ensure that web applications can continue to evolve and become more innovative, while maintaining the web's tradition of openness and preference for standards-driven development.

Although we've been looking at HTML5 features for supporting rich UI applications, HTML5 can also support the development of social and collaborative applications. In the next chapter, you'll look at creating chat message and project planner applications. These apps will teach you about the many new messaging features in HTML5, including cross-domain messaging, WebSockets, and server-sent events (SSE).

## Chapter 4 at a glance

| Topic | Description, methods, and so on | Page |
|---|---|---|
| Server-sent events | Creating events in the browser from the server: | |
| | ■ Creating an `EventSource()` | 111 |
| | ■ Listening to server events with `addEventListener()` | 111 |
| WebSockets | Two-way, event-driven communication: | |
| | ■ Writing applications using `WebSockets` | 116 |
| | ■ Messaging on the client side | 125 |
| Cross-document messaging | Communication between scripts in different windows: | |
| | ■ Sending messages with `postMessage()` | 126 |
| | ■ Receiving messages with `onmessage()` | 126 |

Look for this icon Core API ➡ throughout the chapter to quickly locate the topics outlined in this table.